Line drawings from 3D models: a tutorial

Pierre Bénard, Aaron Hertzmann

LINE DRAWINGS FROM 3D MODELS: A TUTORIAL

PREPRINT

Pierre Bénard
LaBRI (UMR 5800, CNRS, Univ. Bordeaux)
Inria Bordeaux Sud-Ouest
pierre.benard@labri.fr

Aaron Hertzmann
Adobe Research
hertzman@dgp.toronto.edu

ABSTRACT

This tutorial describes the geometry and algorithms for generating line drawings from 3D models, focusing on occluding contours.

The geometry of occluding contours on meshes and on smooth surfaces is described in detail, together with algorithms for extracting contours, computing their visibility, and creating stylized renderings and animations. Exact methods and hardware-accelerated fast methods are both described, and the trade-offs between different methods are discussed. The tutorial brings together and organizes material that, at present, is scattered throughout the literature. It also includes some novel explanations, and implementation tips.

A thorough survey of the field of non-photorealistic 3D rendering is also included, covering other kinds of line drawings and artistic shading.

CONTENTS

1 INTRODUCTION

Humans have been drawing pictures since the days of prehistoric cave painting. Various forms of line drawing have been developed since then, including Egyptian hieroglyphs, medieval etching, and industrial-era printmaking. Nowadays, line drawing and outline drawing methods are used throughout cartoons and comics, architectural rendering, instructional tutorials, and many other settings. Drawing is the starting point for many kinds of tasks, for everyone from children making pictures to professional architects sketching ideas. Drawing seems to be fundamentally connected to how we represent the world visually.

While most computer graphics focuses on realistic visual simulation, over the past few decades, line drawing algorithms have also matured. We now have the ability to automatically create reasonable line drawings from 3D geometry, much like photorealistic rendering. These algorithms provide deep insight into the geometry and topology of line drawings, which can be surprisingly subtle, given how simple line drawing might seem. Versions of these algorithms have been used throughout art, entertainment, and visualization. User evaluation has shown that these algorithms, indeed, accurately describe important aspects of how artists draw lines. This shows that these algorithms can contribute to a scientific understanding of art.

This tutorial provides a detailed guide to the mathematical theory and computer algorithms for line drawing of 3D objects. We focus on the curves known as *occluding contours* or, simply, *contours*. These are the most important curves for line drawing of 3D surfaces. They have a rich theory around them, and, once one understands this theory, understanding how other curves operate is much simpler. We describe the different algorithms required to compute and render these curves, together with references to the literature. We also explain boundary curves and surface-surface intersection curves, since these are straightforward to include and often important. We also discuss open research problems in contour rendering.

In addition, we survey of other topics in 3D non-photorealistic rendering, with extensive pointers to the literature, including: other types of curves, stroke rendering, and non-photorealistic shading. We do not cover the complementary topic of image-based non-photorealistic rendering; for a survey of image-based methods, we refer the reader to the book by Rosin and Collomosse (2013).

The theory of line drawing is currently scattered about and incomplete in research papers. The algorithms for line drawing include many subtleties that are not described in the literature, and many pitfalls await the coder attempting them. There remain some important

open problems, but these gaps are not obvious from the literature. This tutorial is meant to address these issues.

We believe that these topics ought to be known by anyone interested in understanding the curves in visual representational art. It is one where computer graphics can make a unique contribution. Arguably, the algorithmic simulation of line drawing is a crucial step in understanding visual art.

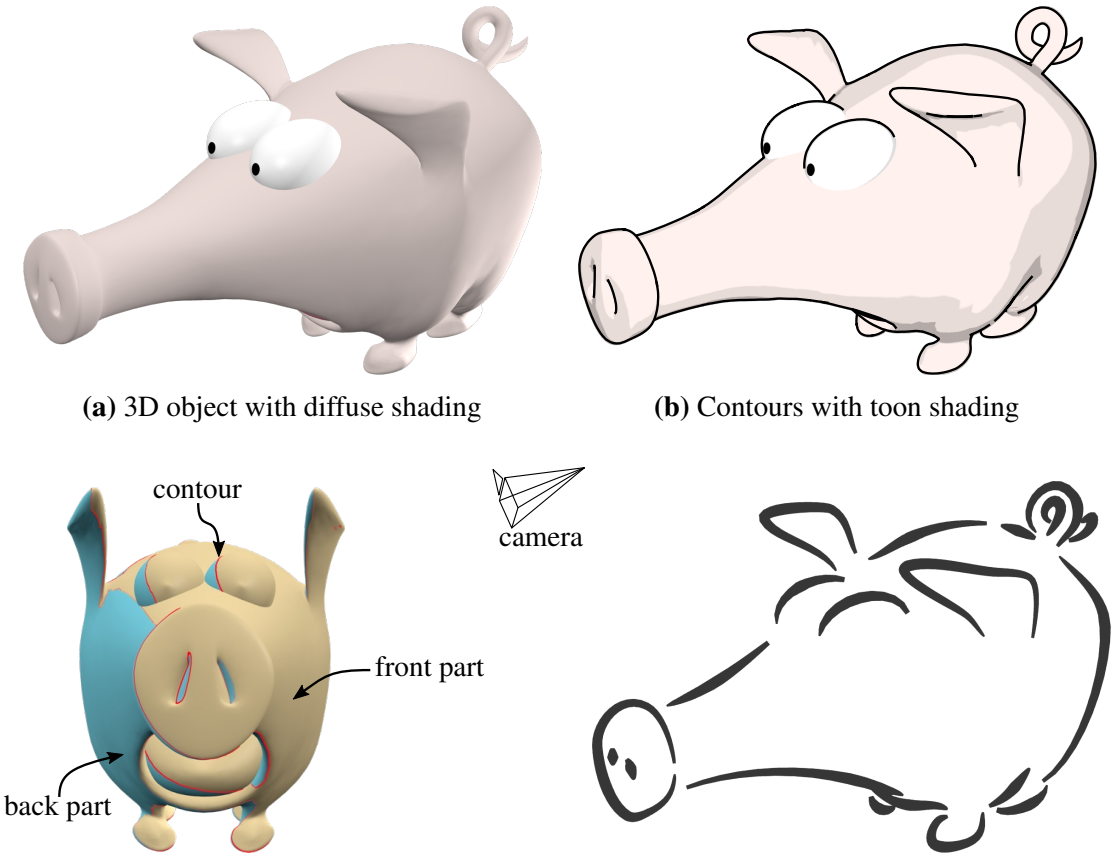
1.1 Occluding contours

This tutorial focuses on the curves known as *occluding contours* or, simply *contours*. In some computer graphics research, these have been called *silhouettes*, though the *silhouettes* are technically a separate set of curves, as we will describe below.

The occluding contours of a simple 3D object are shown in Figure 1.1. As a first definition, suppose we have a 3D object that we wish to render from a specific viewpoint. The occluding contours are *surface curves that separate visible parts from invisible parts*. By rendering the visible portions of these 3D curves together with the object, we get a basic line drawing (Figure 1.1b).

There are many different contour detection and rendering algorithms, and some significant tradeoffs between them. The most important tradeoff is between simple algorithms that produce approximate results, and more complex algorithms that give more precision, control, and stylization capabilities. Just rendering reasonable-looking contours as solid black lines is very straightforward for a graphics programmer. These most basic algorithms can be implemented in a few additional lines of code in an existing renderer, and have been implemented in many real-time applications, including many popular video games (one of the earliest was *Jet Set Radio* in 2000). However, if we wish to stylize the curves, for example, by rendering the curves with sketchy or calligraphic strokes (Figure 1.1d), things become more difficult. With a bit of perseverance, renderings with distinctive and lovely styles can be created. At times, these renderings may still contain topological artifacts that are not suitable for very high-end production. High-quality algorithms that remove these artifacts are more complex; in the extreme, no provably-correct algorithm exists for this problem. However, there are a number of partial solutions that are good enough to be used in many circumstances, and we discuss in detail the issues involved.

Note that, formally, the occluding contour and occluding contour generator are separate curves in 2D and 3D. However, we will frequently use the term “contour” to refer to each of them, since the correct terms are very cumbersome, and the meaning of “contour” is usually obvious from context.



(a) 3D object with diffuse shading

(b) Contours with toon shading

(c) Side view of the scene

(d) Stylized curves

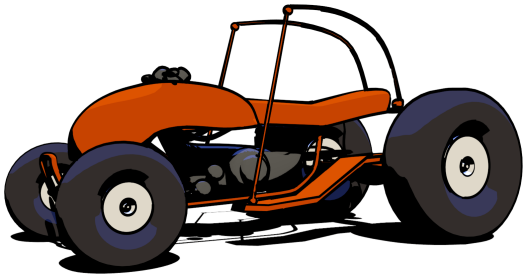
Figure 1.1: Occluding contours — The occluding contours of the 3D model "Origins of the Pig" © Keenan Crane, shown in **(a)** with diffuse shading, are depicted in **(b)** composited with toon shading to produce a cel-like drawing. As illustrated in **(c)** from a side view, they delineate the frontier between the front and back parts of the surface when seen from the camera. These contour curves can be further process to produce stylized imagery, such as the calligraphic brush strokes in **(d)**.

1.2 How to use this tutorial

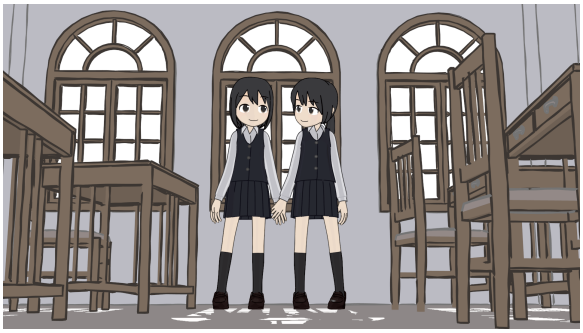
This tutorial is two things: a detailed tutorial of the core contour algorithms, and a high-level survey of nearly all of 3D non-photorealistic rendering. We cover some core topics more thoroughly than any previous publication, and, for other topics, we mainly provide pointers to the literature.

Hence, reading the tutorial directly will give a good overview of the field, but one may skim through survey sections. Alternatively, a practitioner may wish to jump directly to the algorithms relevant to their task.

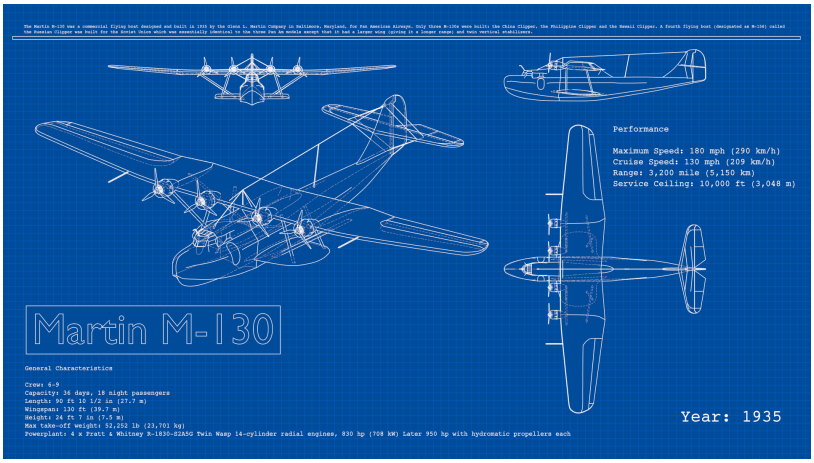
Generally speaking, real-time image-based methods, especially based on graphics shaders, offer the best real-time performance and have been used in many games. These



(a) Buggy by Rylan Wright CC BY



(b) Anime by mato.sus304 CC BY SA



(c) Martin M-130 blueprint by LightBWK CC0

(d) Ryner by Lucas Gogol CC0

Figure 1.2: Artworks created by artists using Blender Freestyle — Each of these is a non-photorealistic rendering, using the techniques described in this tutorial in different ways.

are described in the next Chapter, and pointers to further reading are provided there. This chapter can also help build intuitions for all readers.

The core chapters of the tutorial focus on contour detection and visibility on 3D models. We start with 3D mesh representations, and then apply the same ideas to different smooth surface representations in the subsequent chapters.

We then cover the core topic of detecting contours on meshes (Chapter 3) and computing their visibility (Chapter 4). Contour detection and visibility on meshes is the most basic and well-understood problem, and we go into the most detail in algorithms here. We describe fast, approximate hardware based visibility in Chapter 5.

While it may be tempting to use mesh algorithms for smooth surfaces, in Chapter 6, we explain some of the problems with doing so. We then describe a method called Interpolated Contours that provides a compromise position, being almost as simple as mesh contours to implement, with relatively few inconsistencies.

We then discuss true contours on parametric surface representations (Chapter 7). Understanding these curves involves some differential geometry (reviewed in Appendix A), and the resulting mathematics and theory is rather elegant. We describe detection and visibility algorithms, which are adapted from the mesh algorithms. We describe the different strategies that have been applied to this problem and how they compare. In the following chapter, we then discuss these algorithms as applied to implicit smooth surface representations (Chapter 8).

Finally, we discuss stylized rendering and animations algorithms (Chapter 9), and conclude with a discussion of the state of research and applications in 3D non-photorealistic rendering (Chapter 10).

1.3 The importance of visualizations

Although we have done our best to explain contours in text, they can take some time to wrap your head around. Understanding how the 2D curves and 3D curves relate in an image like Figure 1.1 can be challenging. It is worthwhile spending time with these figures, perhaps starting with simpler examples like different views of a torus, to understand how the 2D and 3D shapes relate, what the curves look like at singular points, and so on.

We provide an interactive viewer at `https://benardp.github.io/contours_viewer/`. Experimenting with this viewer can help give intuitions on contours.

Even better is to use, or build, a 3D visualization. If you implement a 3D contour rendering system, it is essential to also implement visualizations that let you zoom into the 2D drawing and rotate around the 3D model. In each view, you should be able to render the different types of curves and singularities and their attributes. These visualizations are essential for deep understanding of these curves, as well as for debugging and algorithm development. You can start with simple 3D drawings, e.g., rendering contour edges on the

3D model, and coloring mesh faces according to facing direction as in Figure 1.1c. As your system becomes more sophisticated, you may eventually have visualizations like those in Figure 4.6.

These visualizations are also useful in making certain design choices. As we discuss, there is no current foolproof system for smooth contour rendering, and so there are some choices to be made, e.g., selecting heuristics. Good visualizations can also be helpful in understanding how different heuristics behave.

1.4 The science and perception of art

The algorithms described in this tutorial provide a new level of insight and understanding into the science of art (Hertzmann, 2010). For centuries, artists, historians, philosophers, and scientists have sought a formal understanding of visual art: how do we make it, and how do we perceive it? One of the first generative tools in art was the development of linear perspective during the Italian Renaissance. The theory of occluding contours, which is the main subject of this tutorial, originated in perceptual psychology and computer vision, and was developed into the sophisticated algorithms we described here by computer graphics researchers.

As perceptual psychology has developed, so have perceptual theories of art. For example, one of the most influential modern writers on visual art is Ernst Gombrich (1961), who argued that artists created artistic styles of depiction over the centuries. Nelson Goodman (1968) took this further to argue that all artistic style functions purely as a denotational system of symbols, like characters on a written page, and, presumably, purely learned as a product of culture. Rudolf Arnheim (1974) attempted to formulate Gestalt-like perceptual rules to drawings. Sayim and Cavanagh (2011) attempt to apply modern neuroscience to understanding the perception of art.

In attempt to formalize the description of styles, John Willats (1997) created a denotational semantics to describe different kinds of realistic styles — expanded by Willats and Durand (2005) to include insights from computer graphics.

Non-photorealistic rendering provides a generative theory for how artists create representational art. Like any theory, it does not cover every case or describe every phenomenon accurately, nor does it say anything about cultural, psychological, or other outside factors in the work. However, this generative theory provides considerable potential insight into how art is made.

We can compare the generative theory to the world before and after Newtonian mechanics. Before Newton, philosophers like Aristotle could make qualitative observations about how objects move (e.g., “heavy objects like to fall”) but could make no real predictions. Newtonian mechanics is predictive, it generates insights, and leads to real understanding. Likewise, understanding the generative model of representational art provides a potentially compact way to understand many phenomena.

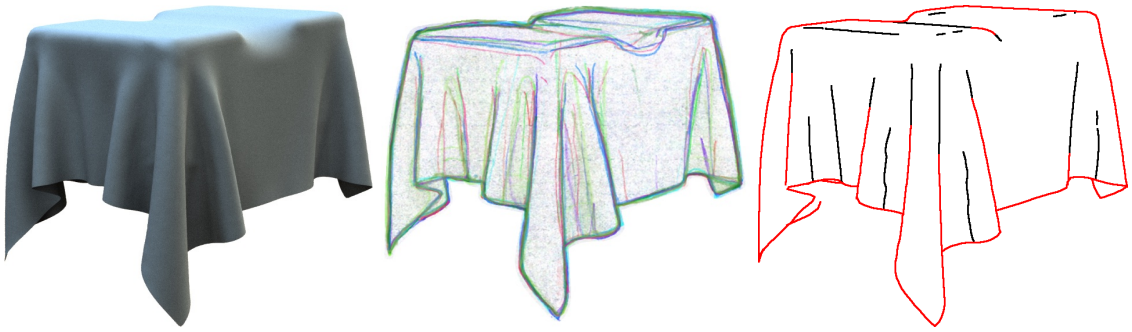


Figure 1.3: Correlation between hand-drawn lines and contours — A 3D model rendered from a given viewpoint and illumination (left) has been hand-drawn by ten artists (center). Observe how consistent the drawings are, especially near the contours of the shape. The contours (in red) and suggestive contours (in black) extracted from the 3D model are depicted in the right. Images taken from the "Javascript Drawing Viewer"[1] of Cole et al. (2008).

Two landmark studies validate and justify the use of line drawing algorithms developed in the non-photorealistic rendering literature. Cole et al. (2008) undertook a careful study of how artists depict 3D objects. They asked a collection of art students to illustrate several 3D models with line drawings, and compared how the artists' drawings related to the line drawing algorithms in this section (Figure 1.3). They showed that roughly 80% of a typical drawing could be explained by existing algorithms. This study helps show which of these algorithms are most useful, while also highlighting gaps in the literature. In a follow-up paper, Cole et al. (2009) showed that line drawing algorithms are also very effective at conveying 3D shape.

1.5 Survey of feature curves

This section surveys other important types of surface curves for line drawing, together with pointers to the relevant literature. The remaining chapters focus solely on contour, boundary, and surface-intersection curves.

Most of these curves have been developed both for artistic use and for visualization purposes (Lawonn et al., 2018). However, some types of curves, such as ridges and valleys, seem useful for visualization without mimicking conventional artist curves as well.

Visibility-indicating curves. *Contours* indicate where parts of the surface become visible and invisible, and also indicate where visibility changes. There are a few other important curves that are important for similar reasons.

Boundary curves are simply the boundaries of the surface. Closed surfaces do not have boundaries. These curves are usually rendered when visible. Boundary curves can indicate

[1] http://gfx.cs.princeton.edu/proj/ld3d/lineset/viewer/index.html

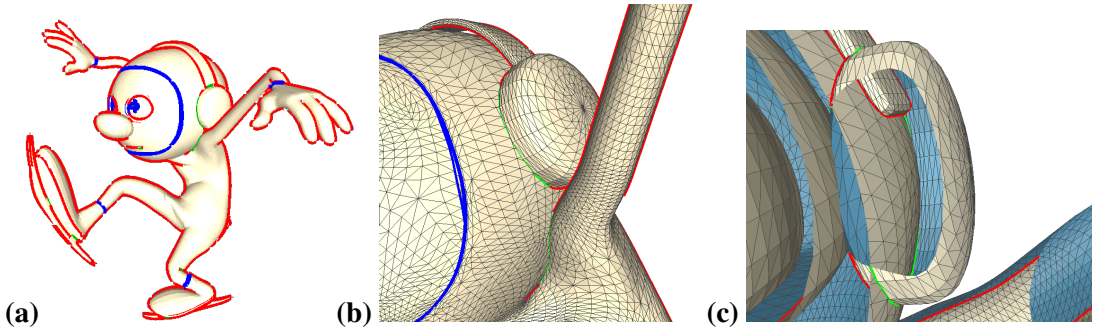


Figure 1.4: Surface-surface intersection curves (from Bénard et al. (2014)) — Professionally-modeled surfaces include many intersections between surface, such as this ice-skating character. Surface intersection curves are shown in green, occluding contours in red, and boundaries in blue. Observe how the ear muffs intersects the headband and the hoodie; the shoulder also happens to intersect the hoodie in this animation frame. **(a,b)** Original surface. **(c)** Cross-section from a different viewpoint. ("Red" character created at Pixar by Andrew Schmidt, Brian Tindall, Bernhard Haux and Paul Aichele, based on the original design of Teddy Newton.)

change of visibility for curves that they intersect in image space, so they are important to handle, and we include them in the discussions of our algorithms in this tutorial.

Surface intersection curves occur when two different sections of surface intersect. These do not occur in the clean models often used in computer graphics research. However, in professional 3D computer animation applications, modelers frequent connect different object parts this way (Figure 1.4). These curves can be detected with standard computer graphics algorithms, and are important to extract since they can indicate changes of visibility with curves that they intersect on the surface. We do not discuss them any further in this tutorial.

All other curves are essentially surface "decorations"; computing them is optional for visibility computations. They typically visualize the surface curvature rather than its outlines.

Contour generalizations. Perhaps the next most significant set of curves are those that generalize contours. These curves were first introduced by DeCarlo *et al.* (DeCarlo, Finkelstein, Rusinkiewicz, and Santella (2003); DeCarlo, Finkelstein, and Rusinkiewicz (2004)), who described a mathematically-elegant generalization of contours and the algorithms needed to render them. Several other variants inspired by this idea were proposed, including apparent ridges (Judd et al., 2007).

The abstracted shading method (Lee et al., 2007) demonstrated how these and lighting-based variants could be computed in image-space. Other variants based on image-space processing include Laplacian Lines (Zhang et al., 2009) and DoG lines (Zhang *et al.* Zhang, Xia, Ying, He, Mueller-Wittig, and Seah (2012); Zhang, Sun, and He (2014)). In addition to speed, image-space lines have the advantage that they automatically remove clutter as a function of image-space line density, although, like all image-based methods, they potentially lose some fine-scale precision and control.

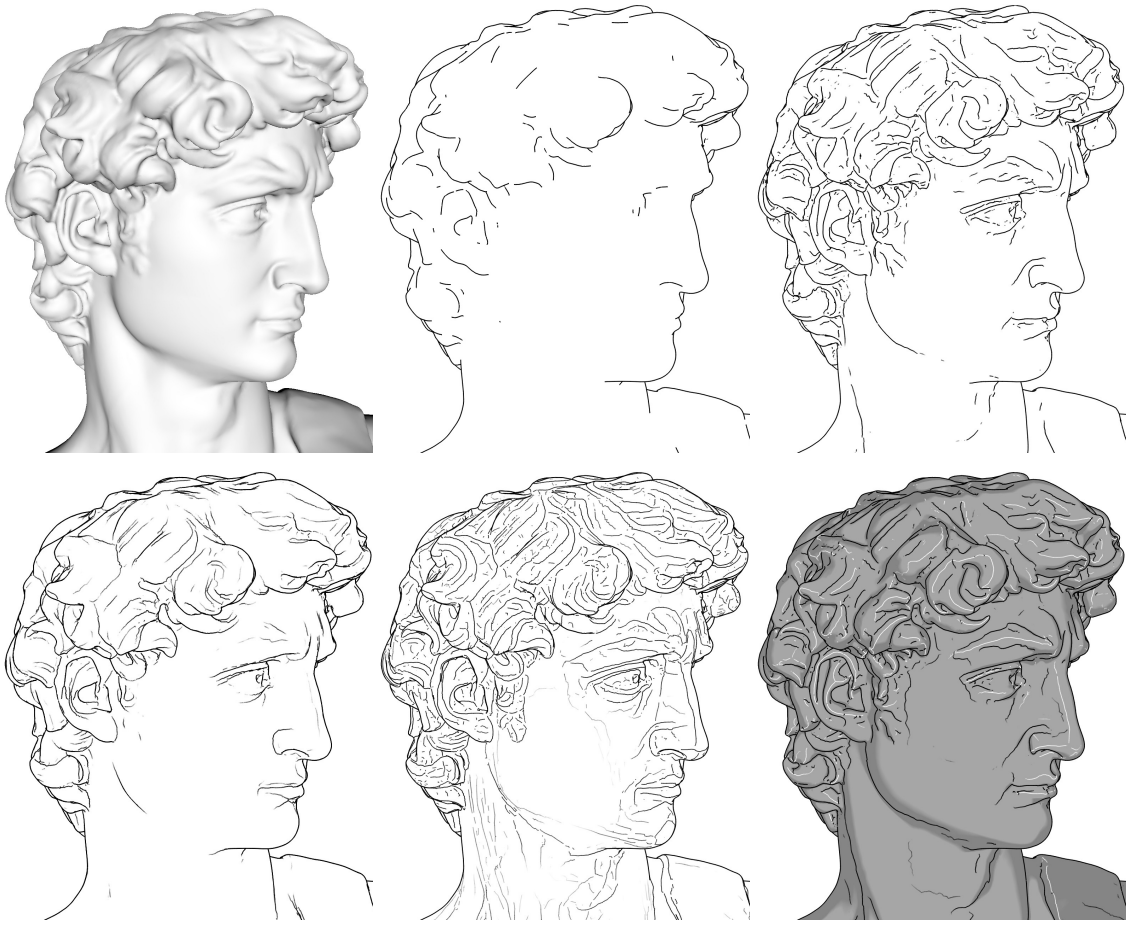


Figure 1.5: Feature curve examples — From left to right, top to bottom: diffuse rendering of the 3D scanned David model by "Scan The World" (http://mmf.io/o/2052), occluding contours (OC), OC + suggestive contours (SC) (DeCarlo et al., 2004), OC + apparent ridges (Judd et al., 2007), OC + ridges & valleys (Rusinkiewicz, 2004), and OC + SC + principal highlights + toon shading (DeCarlo and Rusinkiewicz, 2007). Images generated with "qrtsc" (Cole et al., 2011).

Figure 1.5 shows some examples of these contour generalizations. Including some form of these curves seems essential for capturing how artists depict surfaces; these curves were essential in the study of Cole et al. (2008).

These curves have also been generalized to include highlights that illustrate shading on an object (DeCarlo and Rusinkiewicz, 2007). DeCarlo (2012) provides a thorough survey and comparison of these different types of contour generalizations.

Surface features/properties. Some intrinsic properties of the surface can be drawn, such as sharp creases on smooth surfaces (Saito and Takahashi, 1990), as well as changes in shading (Xie et al., 2007). When objects have assigned texture and materials, one may wish to draw the material boundaries or the texture itself.

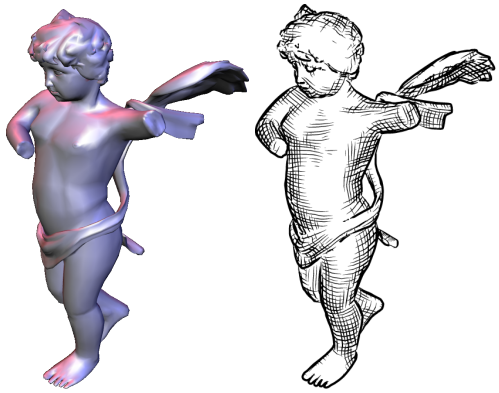


Figure 1.6: Hatching — 3D Cupid model and hatching result obtain with the method of Hertzmann and Zorin (2000).

Hatching. Hatching strokes illustrate surface shape in line drawings. Winkenbach and Salesin (1994) use manually-authored hatching textures and orientations. For more automation, one can use the iso-parametric curves of parametric surfaces (Elber, 1995a; Winkenbach and Salesin, 1996). However, these lines depend on how the shape was authored, and do not generalize to other types of surfaces. Elber (1998) explored many different possible hatching directions, including principal curvature directions, texture gradients, and illumination gradients. Principal curvature-based hatching is supported by perceptual studies suggesting that human perceive hatching strokes as curvature directions (Mamassian and Landy, 1998). Hertzmann and Zorin (2000) refine principal curvature hatching for umbilic regions (Figure 1.6). Singh and Schaefer (2010) describe hatching strokes that follow shading gradients. Since artists draw different types of hatching curves in different situations, Kalogerakis et al. (2012) combine these ideas, describing a machine learning system for learning hatching directions, identifying which hatching rules are used in which parts of a 3D surface. Gerl and Isenberg (2013) additionally offer interactive tools to let the user dynamically control the placement and orientation of hatches.

Surface extrema. Extremal curves, such as ridges and valleys, generalize the notion of ridges and valleys in terrain maps, identifying curves of locally maximal or minimal curvature. These types of curves are a visualization technique that can be useful in understanding surface shape; they do not typically correspond to artist-drawn curves otherwise.

Numerous algorithms have been developed to extract ridges and valleys from various types of geometric models (Interrante et al., 1995; Thirion and Gourdon, 1996; Pauly et al., 2003; Rusinkiewicz, 2004; Ohtake et al., 2004; Yoshizawa et al., 2007; Vergne et al., 2011). A variant, called Demarcating Curves (Kolomenkin et al., 2008; DeCarlo, 2012) can help visualize shapes of different regions on a surface.

1.6 Brief history of 3D Non-Photorealistic Rendering

The earliest 3D computer graphics algorithms were hidden-line rendering algorithms (Roberts, 1963), including methods that we discuss in this tutorial (Appel, 1967; Weiss, 1966). While the mainstream of computer graphics focused on photorealistic imagery, a few works aimed at adding artistic stroke textures to architectural drawings and technical illustrations[2], e.g., (Dooley and Cohen, 1990; Yessios, 1979); meanwhile a number of 2D computer paint programs were developed as well. Many of these papers argued for the potential virtues of hand-drawn styles in technical illustration.

In 1990, the flagship computer graphics conference SIGGRAPH held a session entitled "Non Photo Realistic Rendering," which seems to be the first usage of this term. In this session, two significant papers for the field were presented. Saito and Takahashi (1990) introduced depth-buffer based line enhancements (Chapter 2), which started to create cartoon-like renderings of smooth objects by emphasizing contours and other feature curves. Haeberli (1990) introduced a range of artistic 2D image-processing effects; these papers together demonstrated a significant step forward in the quality and generality of non-photorealistic effects.

Winkenbach and Salesin (1994) demonstrated the first complete line-drawing algorithm from 3D models, including contours and hatching. Their work was seminal in that their method automatically produced beautiful results from 3D models; one could, for the first time, be fooled into thinking that these images were really drawn by hand. Perhaps even more importantly, their work provided a model for one could develop algorithms by careful study of artistic techniques in textbooks and illustrations.

Meier (1996) demonstrated the first research paper focusing on 3D non-photorealistic animation, describing the problem of temporal coherence for animation. Between the beautiful images of Winkenbach and Salesin (Winkenbach and Salesin (1994, 1996)) and beautiful animations of Meier (1996), non-photorealistic rendering was firmly established as an important research direction.

Research activity at SIGGRAPH increased significantly, and the inaugural NPAR symposium on Non-Photorealistic Animation and Rendering met in 2000, sponsored by the Annecy Animation Festival in France and chaired by David Salesin and Jean-Daniel Fekete. Through the following decade, many improvements and extensions to the basic ideas were published, and, occasionally, techniques like toon shading and contour edges appeared in video games and movies. DeCarlo et al. (2003) described Suggestive Contours, which substantially improved the quality of line renderings, while making deep connections to perception and differential geometry, notably the work of Koenderink (1984). Several systems were created to help artists design artistic rendering styles, such as WYSIWYG NPR (Kalnins et al., 2002) and a procedural NPR system called Freestyle (Grabli et al.,

[2]Many works are being omitted from this history. A much more comprehensive bibliography, up to 2011, can be found here: https://www.npcglib.org.

2010). Cole *et al.* (Cole, Golovinskiy, Limpaecher, Barros, Finkelstein, Funkhouser, and Rusinkiewicz (2008); Cole, Sanik, DeCarlo, Finkelstein, Funkhouser, Rusinkiewicz, and Singh (2009)) performed the scientific studies described in Section 1.5 demonstrating that line drawing algorithms were quite good at capturing how artists draw lines.

Since then, research in 3D non-photorealistic rendering has tapered off, despite the presence of several significant open problems. In contrast, interest in image stylization has recently exploded, due to developments in machine learning. Still, 3D non-photorealistic rendering continues to appear in a few games and movies here and there. This tutorial aims, in part, to summarize the field and highlight open problems, to help researchers and practitioners make progress in this field in order to enable them to be more widely used. We discuss future prospects for the field in the Conclusion (Chapter 10).

2 IMAGE-SPACE CURVES

We begin in this chapter by describing simple image-space algorithms. This will provide some informal examples of contours. We will then formally define these curves in the next section.

Most computer graphics rendering systems, including OpenGL, have a way to extract a depth buffer from a rendered scene, such as the one shown in Figure 2.1a. The gray level of each pixel shows the distance of the object from the viewer.

Depth discontinuities in this image correspond to contours. To find them, we can apply an edge detection filter to this image (Saito and Takahashi, 1990), producing the image in Figure 2.1b: these are the occluding contours of the surface. The key assumption of this method is that depth variations between adjacent pixels are small for continuous smooth surfaces, but become large at occlusions. We can also compute a separate normal map image (Decaudin, 1996) shown in Figure 2.1c, and compute its edges, which adds edges at surface creases (Figure 2.1d).

Image-space algorithms work by performing image processing operations on buffers like these ones. They are simple to implement and can run in real-time on graphics hardware. However, they provide limited control over stylization. For example, one cannot easily draw a pencil stroke over the outlines, because there is no explicit curve representation; they are just pixels in a buffer. Furthermore, they can be incorrect, for example, missing contours at small discontinuities or falsely detecting them for highly foreshortened surfaces.

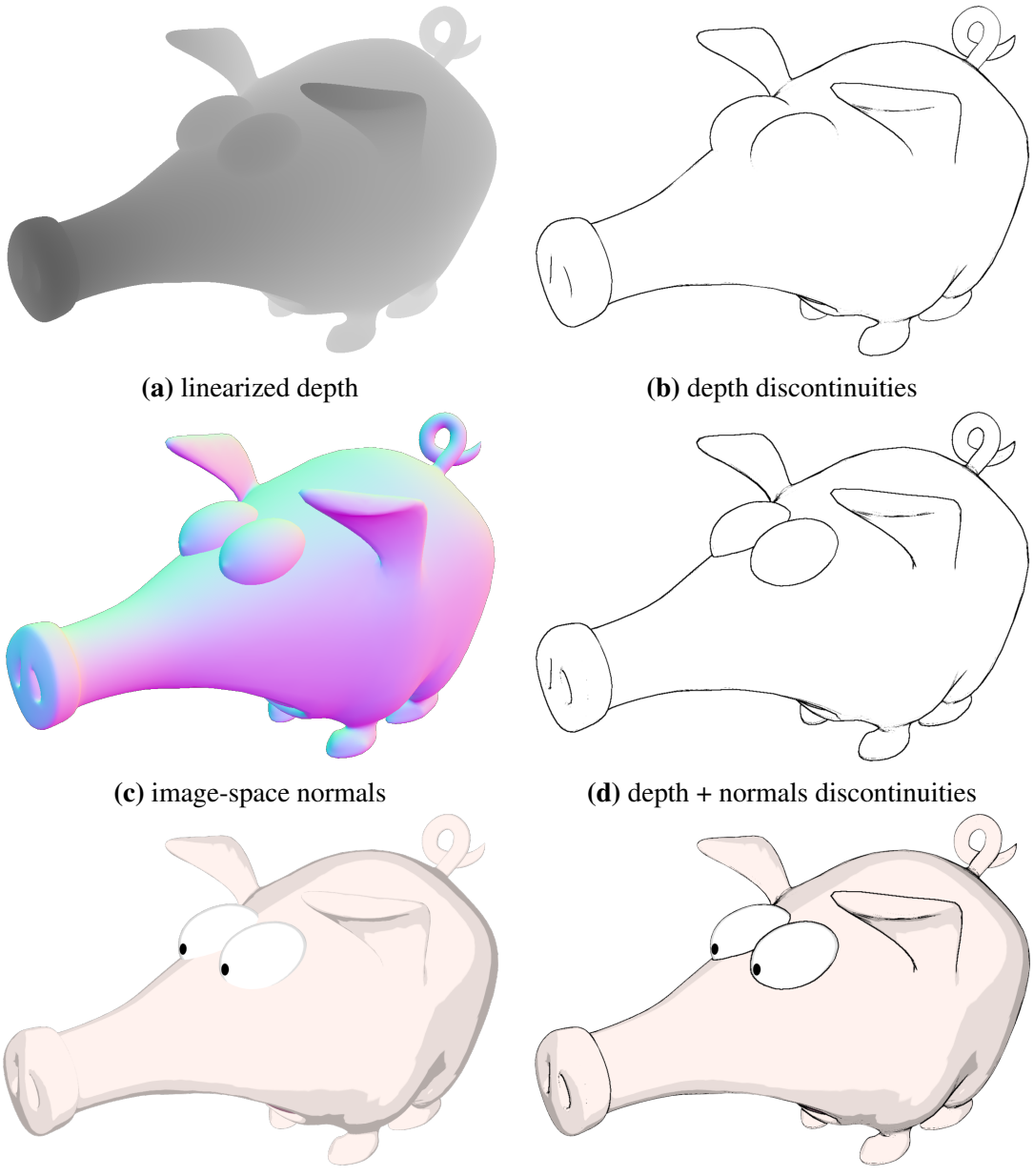
These kinds of edges were used in the video game "Borderlands"; some of the issues involved in getting them to work are described by Thibault and Cavanaugh (2010).

We now describe the depth edge detection algorithm in more detail. A standard choice from image processing is the Sobel filter, which computes approximate depth derivatives (2D gradients) by discrete convolution of the depth buffer D with the following kernels:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

The edge image is then obtained by computing their magnitude:

$$G(x,y) = \sqrt{(D(x,y) \otimes S_x)^2 + (D(x,y) \otimes S_y)^2},$$



(a) linearized depth

(b) depth discontinuities

(c) image-space normals

(d) depth + normals discontinuities

(e) toon shading

(f) egdes composited with toon

Figure 2.1: Image-space edges — The depth buffer of the scene ("Origins of the Pig" © Keenan Crane) is obtained by rasterization and linearized **(a)**, depth discontinuities are then extracted by filtering, here, with a Laplacian of Gaussian filter **(b)**; normals discontinuities can also be considered **(c)** to extract creases; the final edges can eventually be re-composited with the color buffer **(f)**. Images computed with BlenderNPR Edge Node plugin (BlenderNPR, 2015a). The pupils are added separately, as materials on the surface.

and thresholding it by a user-defined threshold τ:

$$Edge(x,y) = \begin{cases} 1 & \text{if } G(x,y) \geq \tau \\ 0 & \text{if } G(x,y) < \tau \end{cases}$$

The results are demonstrated in Figure 2.1. The 3×3 Sobel kernels are the most computationally efficient, but they tend to produce noisy results. As suggested by Hertzmann (1999), they can favorably be replaced by the "optimal" 5×5 kernels of Farid and Simoncelli (1997). Alternatively, second-order derivatives can also be considered, using, for instance, the Laplacian-of-Gaussian filter, or the separable approximation provided by the Difference-of-Gaussians filter (Marr and Hildreth, 1980) and its artistic extensions (Winnemöller et al., 2012).

Note that the depth edge image contains not just contours, but also object boundaries. The normal edge image often includes contours and boundaries, as well as surface-intersection curves. Distinguishing these types of curves, if desired, would be difficult.

As noted by Deussen and Strothotte (2000), GPU depth buffers store non-linear depth values in screen-space, hence depth gradients for remote objects correspond to much larger differences in eye coordinates. If this effect is not desirable, the depth value d ($d \in [0..1]$) first needs to be linearized according to the camera near z_0 and far z_1 clipping plane distances:

$$z = \frac{\frac{z_0 z_1 (d_1 - d_0)}{z_1 - z_0}}{d - \frac{(z_1 + z_0)(d_1 - d_0)}{2(z_1 - z_0)} - \frac{d_1 + d_0}{2}}$$

where d_0 and d_1 are the minimal and maximal values represented in the depth buffer. Alternatively, with modern graphics hardware, the linear camera z-value can directly be written into an offscreen buffer.

2.1 Discussion and extensions

Image-space algorithms only depend on the final image resolution, which is usually an advantage performance-wise, making this approach popular for real-time applications such as games (Thibault and Cavanaugh, 2010). They naturally omit tiny, irrelevant details. However the results may not consistent and predictable when the resolution of the image changes.

Additional buffers. One limitation of depth-buffer algorithms is that they cannot detect edges between objects that are close in depth, such as a foot contact on the ground. However, they can be easily extended to other line definitions by rendering and filtering a *G-buffer* containing, for instance, per-pixel object IDs and surface normals (Figure 2.1c). The former solves the depth ambiguity, whereas first-order normal discontinuities correspond to creases (Saito and Takahashi, 1990; Decaudin, 1996; Hertzmann, 1999; Nienhaus and

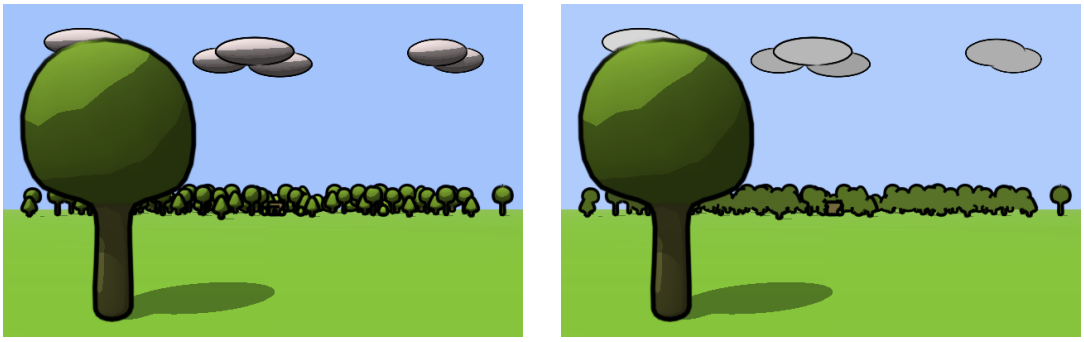


Figure 2.2: Region segmentation (Kolliopoulos et al., 2006) — Toon rendering of a forest scene with no segmentation (left) exhibiting cluttering in the background. With segmentation (right), many of the background trees are grouped together. Contours are only drawn near segment boundaries, resulting in a cleaner image.

Döllner, 2004), and second- and third-order screen-space tensors allow to extract view-dependent ridges and valleys as well as demarcating curves (Vergne et al., 2011).

Kolliopoulos et al. (2006) render scenes by hardware ID buffers to determine pixel-wise object visibility. The scene is adaptively grouped into regions using a segmentation algorithm; the user may determine the grouping parameters so that small objects are grouped together. This is similar to computing planar maps, which will be discussed in more detail in Section 4.8. These planar maps are then stylized in image space (Figure 2.2).

Stroke stylization. These filtering techniques produce a set of disconnected pixels. Hence, modifying the appearance of strokes first requires extracting approximate curves from the buffer. One solution is to fit parametric curves to the edge image using vectorization algorithms (e.g., (Favreau et al., 2016; Bo et al., 2016)), but this tends to introduce inaccuracies and is often too slow for real-time applications.

To create sketchy drawings, Curtis (1998) proposed particles that trace small line segments in the vicinity of the extracted contours. These particles are guided by a density image and a force field which can be obtained by calculating unit vectors perpendicular to the depth buffer's gradient. Although this technique is appealing for its dynamic behavior, the range of style that it can achieve is somewhat limited, and the particle simulation is computationally expensive.

To produce lines of controllable thickness, Lee et al. (2007) proposed to fit a simple analytic profile (degree-2 polynomial) to every pixel of a luminance image, viewed as a height field. This profile locally describes the shape of the closest illumination ridge (or valley). The thickness and opacity of the lines can then be computed based on the distance to the ridge or valley line and its first principal curvature. Vergne et al. (2011) generalized this idea in two ways: first by fitting profiles to various surface features, and then by convolving these profiles with a brush footprint to produce various stylization effects (Figure 2.3). This method achieves real-time performance on the GPU and exhibits a natural coherence in

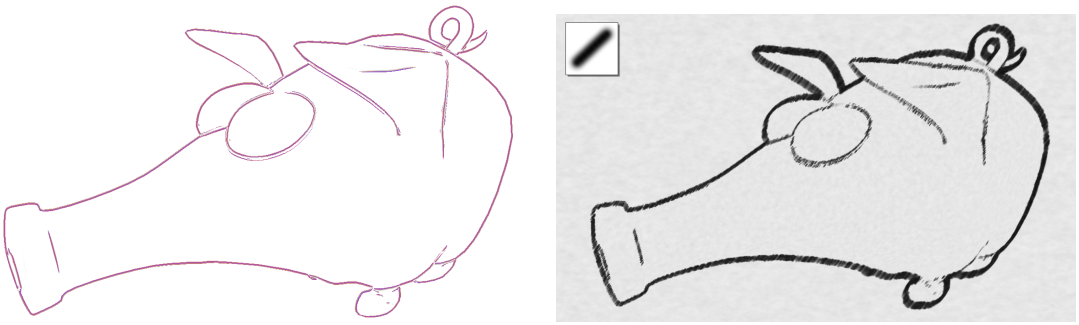
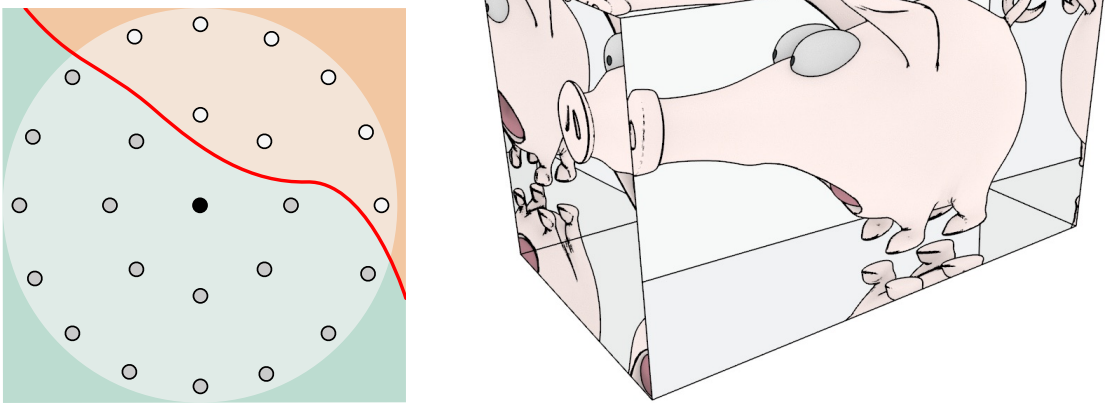


Figure 2.3: Implicit brushes (Vergne et al., 2011) — Surface feature profiles (left) are extracted in image-space and fitted with polynomials; they are then convolved with a brush footprint (inset) to produce stylized lines (right).

animation. However, it is limited to brush styles which are independent of the contour's arc-length; for example, it does not support a brush stroke texture that curves around the object.

Raytracing framework. With a raytracer, a G-buffer can still be computed by casting a ray per pixel and storing the relevant information (e.g., distance to the camera, normal, etc.) at the closest hit point (Leister, 1994; Bigler et al., 2006). To avoid explicitly storing this buffer and allow the user to control the line width, (Choudhury and Parker, 2009) developed a method inspired by cone-tracing. For each per-pixel ray, they sample a set of concentric "probe" rays in an screen-space disc whose radius corresponds to the half line width, that hey call a ray stencil (Figure 2.4a). Then, they compute an edge strength metric based on the proportion of probe samples falling on the same primitive as the central ray. The final pixel color is modulated by this edge factor, producing naturally anti-aliased lines. Ogaki and Georgiev (2018) both simplify and extend this approach to better deal with line intersections, and allow line thickness and color variations. They also support drawing lines in specular reflections and refractions (Figure 2.4b), at the price of storing the tree of reflections and refractions events associated with every pixel ray.



(a) Ray stencil in screen-space (Choudhury and Parker, 2009)

(b) Image generated with Arnold Toon shader (Ogaki and Georgiev, 2018).

Figure 2.4: Ray-traced feature lines — **(a)** Around a central ray (black dot) a stencil of rays (grey and white dots) is cast to estimate the foreign primitive area, i.e., the proportion of samples intersecting a different primitive than the central one (orange vs. green surfaces). **(b)** Image-space depth, ID and normals discontinuities extracted taking into account reflections and refractions.

3 Mesh Contours: Definition and Detection

This chapter formally introduces the occluding contours of polyhedral meshes. We begin with some basic definitions of the mesh and viewing geometry, and then give formal definitions of contours. We then describe a range of extraction algorithms for faster detection. The following chapter will then discuss visibility computations.

Extracting contours from meshes can allow exact computation of the contour topology, allowing for more sophisticated curve stylization algorithms, while also fixing potential problems with the algorithms from the previous chapter.

3.1 Meshes

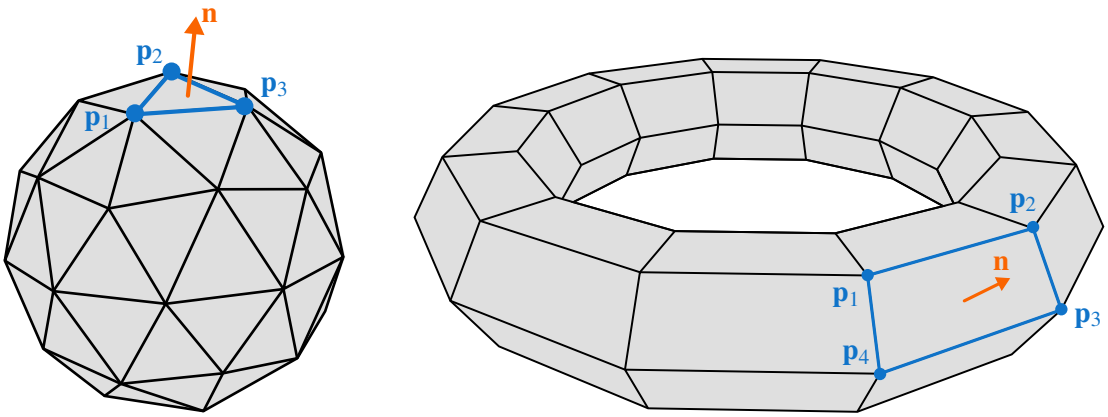
A polyhedral mesh comprises a list of vertices and a list of faces, each face containing three or more vertices (Figure 3.1). Faces meet in mesh edges, each edge connecting two vertices. The mesh *connectivity*, describes the incidence relations among those mesh elements, e.g., adjacent vertices and edges of a face. The mesh *geometry* specifies 3D position of each vertex: $\mathbf{p} = [p_x, p_y, p_z]^\top$. In computer graphics, most polyhedral meshes are either triangular or quadrilateral meshes (Figure 3.1). In this tutorial, we focus on triangular meshes, although the definitions and algorithms presented below generalize to any polyhedral meshes with planar faces. Non-planar faces, such as non-planar quad faces, need to be subdivided into planar faces.

The normal of a face is the vector orthogonal to all edges of the face, which can be computed by the cross-product: $\mathbf{n} = (\mathbf{p}_3 - \mathbf{p}_1) \times (\mathbf{p}_2 - \mathbf{p}_1)$, where $\mathbf{p}_{1:3}$ are any three vertices of the face taken in clockwise order. Note that the normal orientation depends on the order of the vertices (e.g., swapping vertices 1 and 2 would reverse the normal direction). The ordering of the three vertices is usually encoded in the mesh data structure.

We further assume that the mesh is *manifold*, i.e., (1) each edge is incident to only one or two faces, and (2) the faces incident to a vertex form a fan, either open or closed.

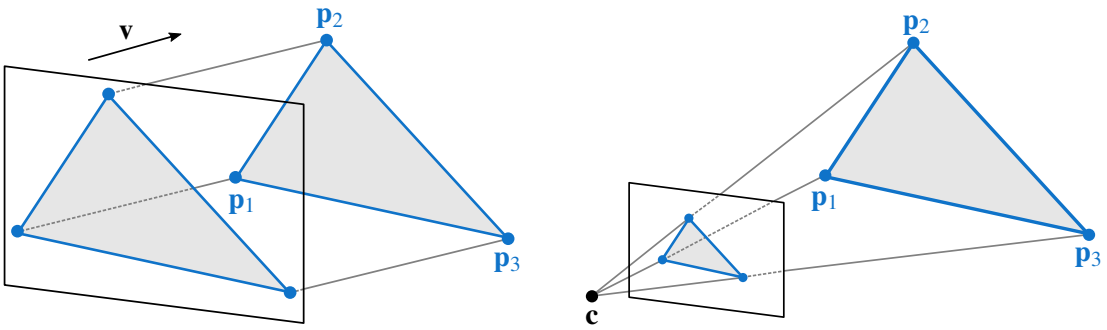
3.2 Camera viewing

The polyhedral mesh will be projected by either orthographic (parallel) or perspective (central) projection. For orthographic projection in the *view direction* $\mathbf{v}$, a given scene



(a) triangle mesh **(b)** quad mesh

Figure 3.1: Polygonal meshes — Since each face of a polygonal mesh is planar, its normal $\mathbf{n}$ can be computed as $(\mathbf{p}_3 - \mathbf{p}_2) \times (\mathbf{p}_2 - \mathbf{p}_1)$, where $\mathbf{p}_{1:3}$ are any three vertices of the face.



(a) orthographic projection **(b)** perspective projection

Figure 3.2: Projections — The triangular face formed by the vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ is viewed **(a)** under orthographic projection along the view direction $\mathbf{v}$, and **(b)** under perspective projection of center $\mathbf{c}$.

point $\mathbf{p}$ is projected to the image plane by intersecting the line that passes through $\mathbf{p}$ in the direction $\mathbf{v}$ — called the the *visual ray* — with the image plane (Figure 3.2a). The point $\mathbf{p}$ is visible if the visual ray does not intersect any other surface point before reaching the image plane; otherwise it is invisible.

For perspective projection, the camera is defined by the position of its center $\mathbf{c}$ and an image plane (Figure 3.2b). In this case, the visual ray is the line from $\mathbf{c}$ to the scene point $\mathbf{p}$; the corresponding view direction $(\mathbf{c} - \mathbf{p})$ is not constant anymore; it depends on $\mathbf{p}$. The projection of $\mathbf{p}$ remains the intersection of the visual ray with the image plane.

The mesh *boundary* is the set of edges where each edge is adjacent to only one mesh face. A surface is *closed* if it has no boundary, otherwise it is *open*.

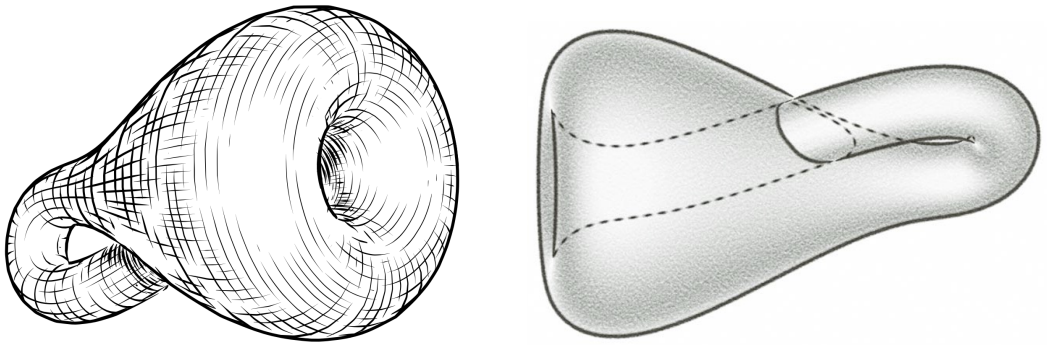


Figure 3.3: Contour renderings of a non-orientable surface, the Klein bottle, left by Hertzmann and Zorin (2000) and right by Kalnins et al. (2003). Right image generated with "Jot" (Kalnins et al., 2007).

3.3 Front faces and back faces

We assume that the mesh is *orientable*. Informally, this requires that all adjacent pairs of faces have consistent normal directions, facing the same side of the surface. This gives the surface a consistent notion of "inside" and "outside", and rules out esoteric surfaces like the Möbius strip and the Klein bottle (Figure 3.3). For an open surface, we can think of the surface as a subset of an orientable closed surface that will only be seen from certain viewpoints.

More formally, orientability can be determined in a mesh data structure as follows. Each triangle in the data structure represents its vertices in a cyclic ordering. Two adjacent faces are consistent if the two vertices of their common edge are in opposite order.

A face is *front-facing* if the camera position lies on the side of the face pointed to by the face's normal, i.e., $(\mathbf{c}-\mathbf{p})\cdot\mathbf{n} > 0$ (Figure 3.4). It is *back-facing* if the camera lies on the other side of this plane. In orthographic projection, a face is front-facing when $\mathbf{v}\cdot\mathbf{n} < 0$.

We assume that only front-faces may be visible; back-faces must always be invisible. This occurs in two ways. First, when a closed mesh with outward-facing normals is viewed from the outside, the back-faces must all be occluded by front-faces. Second, in a professional animation setting, objects are often modeled with open surfaces, but with the camera movements constrained so that only the front facing regions will be visible.

3.4 Mesh contours and boundaries

The general definition of occluding contours, for all surfaces, is as follows.

Definition 3.4.1 (occluding contour generator)**.** *For a given viewpoint, the* occluding contour generator *is a curve on the surface that delineates the frontier between what is locally visible and invisible, that is, between front- and back-facing surface regions.*

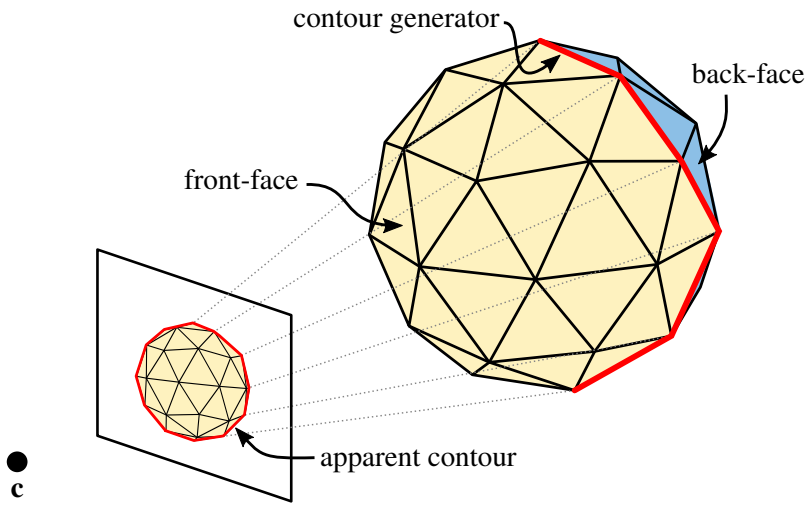


Figure 3.4: Front faces, back faces, and contour — The front faces are shown in yellow, and are visible to the camera. The back faces are in blue, and are not visible to the camera. The contour generator separates the front facing regions from the back-facing regions of the surface. The apparent contour is the visible projection of this curve onto the image plane.

As such, they mark any depth discontinuity either between the surface and the background in the image, or where parts of an object pass in front of itself.

Definition 3.4.2 (occluding contour)**.** *For a given viewpoint, the* occluding contour *(or apparent contour) is the visible 2D projection of the occluding contour generator.*

These definitions, as applied to meshes, are:

Definition 3.4.3 (mesh contour generator)**.** *The collection of all mesh edges that connect front-faces to back-faces are together called the* occluding contour generator *(Marr, 1977). The visible projection of the contour generator onto the image plane is called the* occluding contour*, or,* apparent contour.

Despite the different terminology here, we will often simply use the term "contour" to refer to these different curves, where the meaning is obvious from context, simply because terms like "occluding contour generator" are rather cumbersome.

In the literature, there is considerable variation in how these terms are used. The *silhouette* is the subset of the contour that separates an object from the background behind it. In the computer graphics literature, the word "silhouette" was often used to mean "contour", especially prior to 2003. Koenderink uses the term "rim" to refer to the occluding contour generator. Some authors use the term "contour" to refer to any image curve.

3.5 Generic position assumption

We further assume that the mesh is in *generic position*, which is a helpful trick for avoiding many tedious technicalities.

Loosely speaking, the generic position assumption implies that the mesh does not have any specific "weird" connectivity, nor does the camera's view of the mesh — this frees us from handling many possible special cases.

More precisely, in generic position, any relevant topological properties of the mesh and camera together are robust to infinitesimal perturbations.

For example, it is theoretically possible for a face of the mesh to be exactly edge-on: $(\mathbf{c} - \mathbf{p}) \cdot \mathbf{n} = 0$. This is a face which is neither front-facing nor back-facing. Correctly drawing the contours through this face, with correct contour topology for stylization, would require some extra effort on top of the basic algorithms we will describe. However, if we added an infinitesimal amount of random noise to $\mathbf{c}$ or any of the vertices $\mathbf{p}$, then the face would no longer be edge-on (with probability 1). Other non-generic cases that can cause problems include degenerate edges (adjacent vertices have the exact same coordinates), and coincident geometry (e.g., two distinct triangles lie in the same plane and overlap).

In general, handling non-generic cases like these require extra effort to implement, and they would be largely unenlightening to explain in this tutorial. Even enumerating potential non-generic cases could be quite tedious and difficult. Furthermore, the research literature has largely ignored non-generic cases.

For real-valued geometry that is randomly-positioned, violations of the generic position assumption are zero-probability events. Even in floating-point computations, the odds of the assumption being violated are vanishingly rare. In some cases, genericity violations may be intentional, such as in mechanical illustration and industrial design applications, where edge-on faces are common. For these applications, some specific non-generic cases would need to be handled.

A simple fix to violations of generic position is to randomly add a tiny random number to each vertex coordinate of the mesh; by definition, this will cure all non-generic cases. For example, the edge-on face described above would become either front-facing or back-facing. More principled handling is potentially application-dependent, and we do not discuss it further in this tutorial.

3.6 Contours are sparse

As noted by Markosian et al. (1997); Kettner and Welzl (1997); Sander et al. (2000); McGuire (2004), contour edges only represent a tiny percentage of the total number of mesh edges. For a reasonable polyhedral approximation of a smooth surface, Glisse (2006) showed that the contour length, averaged over all viewpoints, is in the order of $\sqrt{n}$ where n is the number of faces in the mesh. In practice, for general man-made triangular meshes, McGuire (2004) measured empirically a trend closer to $n^{0.8}$. For example, the Buddha model has over 1 million faces but only around 50k contour edges on average from different views. Of these, a large fraction are surely concave, and thus can trivially be marked as always invisible, as discussed in Section 4.2.

Additionally, edges that are more convex are more likely to be contours than edges that are flatter (Markosian et al., 1997). For example, a nearly-flat edge only becomes a contour from a narrow range of camera positions, unlike a very sharply convex edge.

3.7 Extraction algorithms

We now survey different algorithms for detecting the set of contour edges on a mesh. These range from the basic brute-force procedure, to more sophisticated data structures and algorithms. Computing the apparent contours further requires determining the visibility of the contour generators; solutions to this challenging problem will be presented in Chapter 4 and Chapter 5.

3.7.1 Brute force extraction

The basic brute force algorithm directly stems from Definition 3.4.3. For a given viewpoint, the algorithm consists in iterating over every mesh edge, computing the normals of its two adjacent faces, and checking whether their dot-products with the view direction have opposite the signs. For perspective projection, any position on the edge can be used to define the view direction; the first vertex of the edge is commonly chosen.

Representing the mesh with a half-edge data structure (Campagna et al., 1998) makes these operations easier to implement. To avoid redundant calculations, the face normals are usually precomputed and stored as face attributes. The iteration over the mesh edges must be performed every time the camera or object position changes, which is very expensive for complex models. Figure 3.5 shows three results of this algorithm; the hidden contours are illustrated with dotted lines.

3.7.2 Pre-computation for static meshes

In many applications, parts of the 3D scene are static, or rigid. In such a case, a data-structure can be built for each static mesh during an advance pre-process, so that the search is significantly accelerated at rendering time. At run-time, contour extraction can be a function of the number of contour edges, rather than the number of mesh edges, yielding a substantial time savings due to the sparsity of contours (Section 3.6).

Orthographic dual space. For orthographic projection, Benichou and Elber (1999) and Gooch et al. (1999) proposed a dual space for fast contour detection. The dual space is a 3D coordinate system $\mathbf{s} = (s_1, s_2, s_3)$. Each mesh face is mapped to a single point $\mathbf{s}$ in the dual space, with coordinates given by the face normal: $\mathbf{s}_i = \mathbf{n}_i = (n_x, n_y, n_z)$. Likewise, the orientation function, based on view direction $\mathbf{v} = (v_x, v_y, v_z)$, is mapped to a plane in the dual space:

$$g(\mathbf{s}) = \mathbf{v} \cdot \mathbf{s} = v_x s_1 + v_y s_2 + v_z s_3 = 0 \tag{3.1}$$

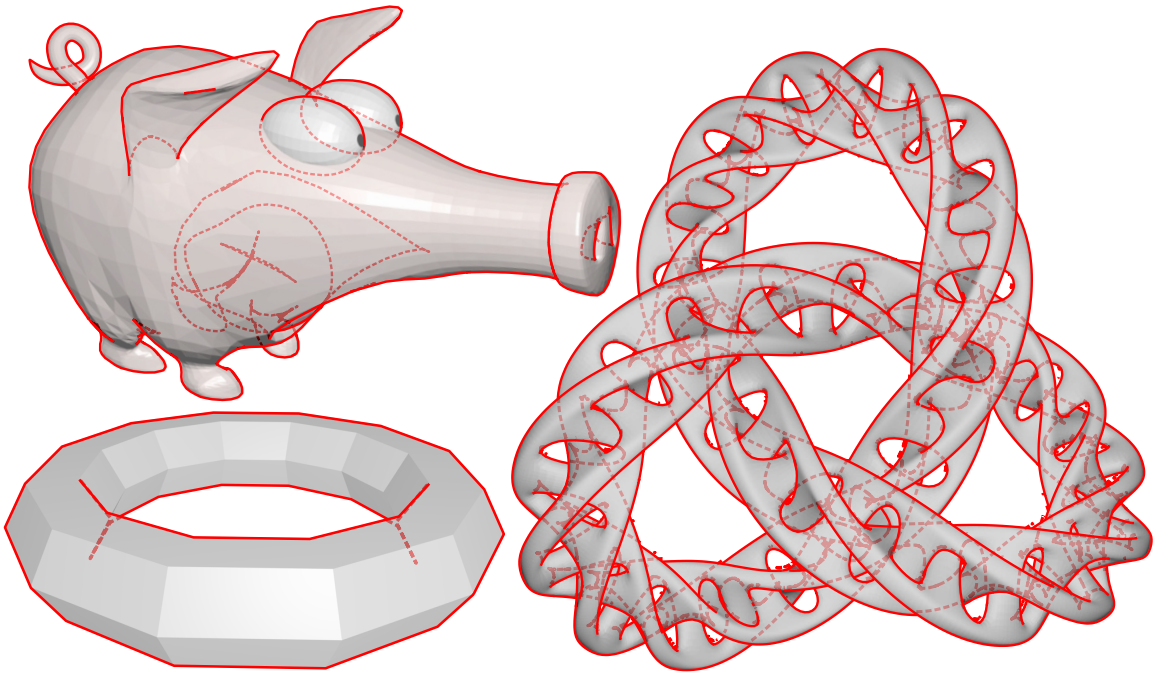
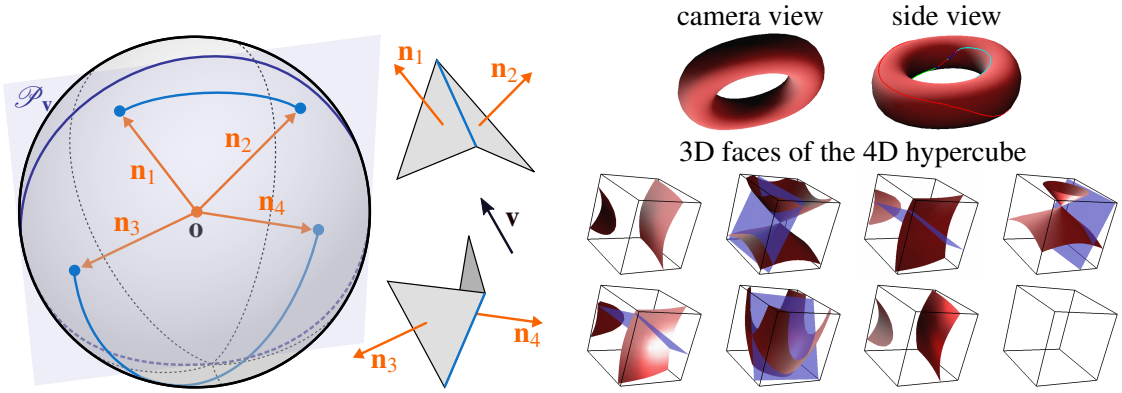


Figure 3.5: Results — Mesh contours extracted from a low-resolution torus (bottom left), the "Origins of the Pig" © Keenan Crane (top left), and the "Moebius Torus Knot" © Francisco Javier Ortiz Vázquez (right). Hidden contours are depicted with dotted lines (visibility algorithms are discussed in the next Chapter).

For front faces, $g(\mathbf{s}) > 0$, and $g(\mathbf{s}) < 0$ for back faces. A mesh edge between faces (i, j) on the original surface corresponds in the dual space to a line segment $\overline{\mathbf{n}_i\mathbf{n}_j}$. In the dual space, when the orientation plane intersects a line segment, this line segment must correspond to a contour edge on the original surface. Hence, contour detection is reduced to intersecting a plane with a set of line segments, which can be accelerated by standard geometric data-structures, such as octrees or BSP trees.

In implementation, the 3D space does not need to be represented; a 2D space is sufficient. Specifically, one can observe that the normals can be arbitrarily scaled without changing the results. Scaling each point to have unit norm projects the points onto the Gaussian sphere, and, within the Gaussian sphere, line segments become arcs (Figure 3.6a). For computation, all points can be projected onto a unit cube (Benichou and Elber, 1999), or a hierarchy of platonic solids (Gooch et al., 1999). Hence, arcs are transformed into line segments, reducing the 3D intersection test to a set of 2D intersection tests, that can be further accelerated with standard 2D data-structures.

Perspective dual space. The above approach can be generalized to perspective projection (Hertzmann and Zorin, 2000). In this case, a 4D dual space is used conceptually, but a 3D dual space is used in practice.



(a) Gaussian sphere **(b)** 4D dual space

Figure 3.6: Dual spaces — Preprocessing by **(a)** projecting the normals of two adjacent faces onto the Gaussian sphere, or **(b)** constructing a representation of the mesh in 4D space based on the position and tangent planes of its vertices. At runtime, finding the contour edges consists in computing the intersection of the dual viewing plane (in blue) with **(a)** circular arcs or **(b)** the dual surface, which can be further accelerated by space-partitioning data-structures.

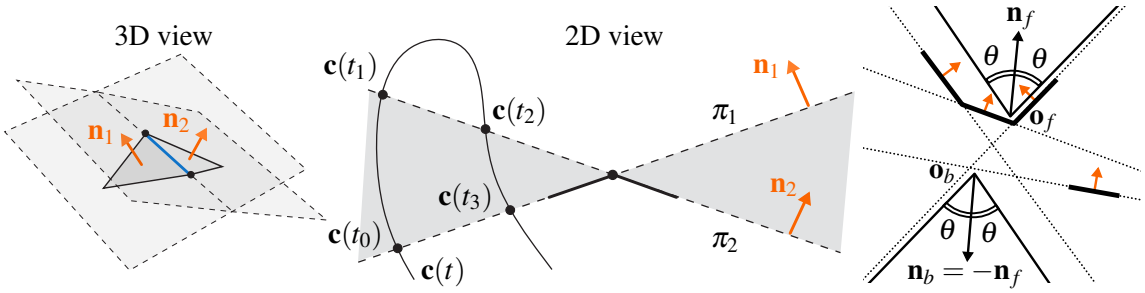
A mesh face with position $\mathbf{p} = (p_x, p_y, p_z)$ and normal $\mathbf{n} = (n_x, n_y, n_z)$ is mapped to a dual point $\mathbf{s} = (s_1, s_2, s_3, s_4) = (-n_x, -n_y, -n_z, \mathbf{p} \cdot \mathbf{n})$. (Any point on the face may be used.) Given the camera center $\mathbf{c}$, the orientation function is mapped to a dual hyperplane:

$$g(\mathbf{s}) = (c_x, c_y, c_z, 1) \cdot s = 0.$$

Hence, front-faces have $g(\mathbf{s}) > 0$ and back-faces have $g(\mathbf{s}) < 0$. A mesh edge between faces i and j corresponds to a dual line segment $\overline{\mathbf{s}_i \mathbf{s}_j}$. Any line segment that intersects the dual hyperplane corresponds to a mesh contour. Hence, finding all contour edges reduces to a 4D hyperplane intersection with a set of line segments. Orthographic cameras can also be handled in this dual space with $g(\mathbf{s}) = [-v_x, -v_y, -v_z, 0] \cdot s = 0$.

As in the orthographic case, the dual points can be scaled arbitrarily without changing the results. Hence, a 3D space can be used. Hertzmann and Zorin (2000) normalize each dual point $\mathbf{s}$ using the l_∞ norm — effectively projecting it on the surface of the unit hypercube. The surface of the unit hypercube can be represented as eight octrees. Each dual point is stored in one of the octrees. At runtime, the viewpoint is converted into a dual plane, and the dual plane is intersected with the eight octrees. The expected complexity is linear to the number of contour edges.

Animation. These data structures can further be exploited to accelerate detection during animation. In dual space representations, when the camera makes small moves, it is possible to only visit a small portion of the dual space. Pop et al. (2001) and Olson and Zhang (2006) describe incremental methods that are able to update an existing set of contour edges when the camera moves.



(a) Time intervals **(b)** Anchor cones

Figure 3.7: Spatial partitioning — **(a)** A given edge is on the contour generator if the viewpoint trajectory $\mathbf{c}(t)$ is inside the intersection (in grey) of one positive and one negative half-space defined by the face supporting plane π_1 and π_2, i.e., during the time intervals $[t_0,t_1]$ and $[t_2,t_3]$ in this example. **(b)** The front and back-facing anchored cones are defined by their center $\mathbf{o}_{f|b}$, an opposite normal $\mathbf{n}_f = \mathbf{n}_b$ and a common half opening angle θ, here visualized in 2D for four oriented segments.

If the viewpoint trajectory is known in advance and can be represented by a polynomial curve $\mathbf{c}(t)$ of degree d, Kim and Baek (2005) showed that there are at most $d+1$ time-intervals $[t_i,t_{i+1}]$ at which an edge can be a contour. Those intervals can thus be pre-computed for each edge, by intersecting the polynomial curve with the supporting planes π_1 and π_2 of the edge's adjacent faces (Figure 3.7a), and stored in an array or a tree data-structure. At runtime, the contour edges can then easily be updated incrementally during the camera motion along the prescribed trajectory. The incremental update mechanism of Pop et al. (2001); Olson and Zhang (2006) is more computationally demanding, but it is not constrained to a fixed camera path.

Cone trees. Sander et al. (2000) proposed accelerating contour extraction using a forest of search trees constructed over the mesh edges. Taking inspiration from previous work on back-face culling, their key idea is to build, for each edge of the mesh, a hierarchy of face clusters. At runtime, clusters whose faces are all front-facing or all back-facing can be fully discarded. To conservatively decide in constant time whether a cluster is front- or back-facing, Sander et al. (2000) compute and store two open-ended anchored cones per cluster: one cone inside which any viewpoint would make the face cluster entirely front-facing, and another cone making the cluster back-facing (Figure 3.7b). They demonstrated that, experimentally, this approach also has linear complexity with respect to the number of contour edges. However, their data structure construction can be extremely slow.

3.7.3 Randomized search

The above data structures are not useful for deforming meshes. The following randomized algorithm, proposed by Markosian et al. (1997), works for any mesh, though it is not guaranteed to detect all edges.

The method first selects a few mesh edges at random. Because contours are sparse, the probability of finding a first contour edge is rather low. However, since mesh contours form continuous chains of edges on the surface, once a first contour edge has been found, spatial coherence can be leveraged to explore adjacent edges in an advancing front manner and trace the full contour loop. By further assigning to each edge a probability inversely proportional to the exterior dihedral angle α (in radians) between its adjacent faces, the chance of finding contour edges is increased since, given a random view direction, the probability that an edge is a contour is α/π. Derivations for this probability can be found in McGuire (2004) (perspective case) and Elber and Cohen (2006) (orthographic case).

In addition, for small viewpoint changes, Markosian et al. (1997) observed that temporal coherence can also be leveraged by re-seeding the search in the new frame from the previous frame's contour, and by searching for contour edges in its vicinity, moving towards (resp. away) from the camera if the edge is adjacent to back-faces (resp. front-faces).

This approach does not guarantee that all contour edges will be found, but it will usually detect the longest mesh contours. If the algorithm samples edges without replacement, then it will converge to the correct solution once it has visited every edge.

4 Mesh Curve Visibility

Once we have found the curves on a mesh, we need to determine which portions of them are visible. In doing so, we will also build a data structure, called the *view graph*, that represents the topology of the visible curves.

This chapter introduces the algorithms used for efficiently computing correct visibility for edges on the surface. This question is related to the more general hidden-line removal problem, which dates back to the earliest ages of Computer Graphics, at the beginning of the sixties. Roberts (1963); Weiss (1966) devised the first known solutions to this problem, using brute-force ray tests. Appel (1967) introduced Quantitative Invisibility as a way to greatly decrease the number of ray tests required, and improve accuracy.

It can be tempting to implement many of these algorithms with heuristics. However, if not implemented carefully, the visibility operations here can be very sensitive. Our goal is to compute global curve topology, and depending on implementation, the visibility of a large curve may depend on a single visibility test somewhere on the curve. If this visibility test is erroneous, an entire curve from the drawing may disappear. Hence, it is important to formulate these algorithms to carefully track curve visibility and topology, rather than using heuristics. Even with mathematically correct operations, numerical instability can also cause errors. Techniques for robust visibility computation are discussed in Appendix C.

4.1 Ray tests

For a perspective camera, a point $\mathbf{p}$ on the surface is visible from the camera center $\mathbf{c}$ if the line segment $\overline{\mathbf{pc}}$ intersects the image plane and does not intersect any other surface point (Figure 4.1). Determining visibility this way is called a *ray test*, since it amounts to casting a ray from $\mathbf{c}$ and checking if the tripling is the first object hit. (For an orthographic camera, the test involves a line segment from $\mathbf{p}$ to the camera plane along the ray $-\mathbf{v}$.) Ray tests can be accelerated by spatial subdivision data structures, such as a 3D grid or a bounding volume hierarchy (Pharr et al., 2016, Chapter 4).

In principle, the apparent contour could be rendered by separately testing the visibility of many points on the contour generator, and connecting the visible points. However, as noted by Appel (1967), this would be both computationally expensive, because it would require testing a large number of points between which the visibility does not change, and inaccurate, since it would miss the points where curves transition between visible and invisible. Instead, we will use techniques to propagate visibility on the surface.

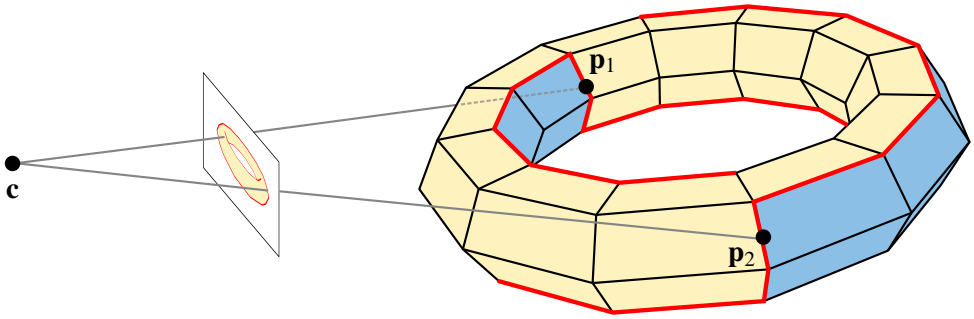


Figure 4.1: Visibility, ray tests, and convex/concave contours — A point is visible if the line segment from the camera to the point does not intersect any other surface point. Determining this is called a ray test. In the example here, the segment from $\mathbf{p}_1$ to the camera $\mathbf{c}$ intersects another part of the surface, so $\mathbf{p}_1$ is not visible. The segment from $\mathbf{p}_2$ does not intersect the surface so it is visible. One can avoid computing one of these ray tests: $\mathbf{p}_1$ lies on a concave contour point, so it must be invisible. The other point, $\mathbf{p}_2$, is on a concave contour, so a ray test is necessary to determine if it is visible.

4.2 Concave and convex edges

We can classify mesh edges as to whether they are concave and convex, which provides an additional visibility constraint, and will be helpful for identifying singularities in the next section (Markosian et al., 1997). The content of this section is new for this tutorial, building on (Markosian et al., 1997; Koenderink, 1984).

A mesh edge is concave if the angle between the front-facing sides of its two faces is less than π. It is convex if the angle is greater than π. (Note that this is the angle on the outside of the surface, and so it is different from the dihedral angle.) Equivalently, when the edge is convex, each face is on the back-facing side of the other face.

Contours on concave edges must always be invisible: if a concave edge is viewed at a grazing angle — where a contour appears — the edge is hidden inside the surface from that viewpoint. Only convex edges can produce visible contours. Figure 4.1 shows examples of convex/concave edges, and how they can be invisible or visible. We provide a new, formal proof of this in Appendix B.

As a result, ray tests are never necessary for contours on concave edges. Additionally, for static surfaces, concave edges can be omitted from any detection data structure (Section 3.7.2), if hidden lines will not be rendered.

Algorithms for determining whether an edge is convex or concave are given in Appendix C.

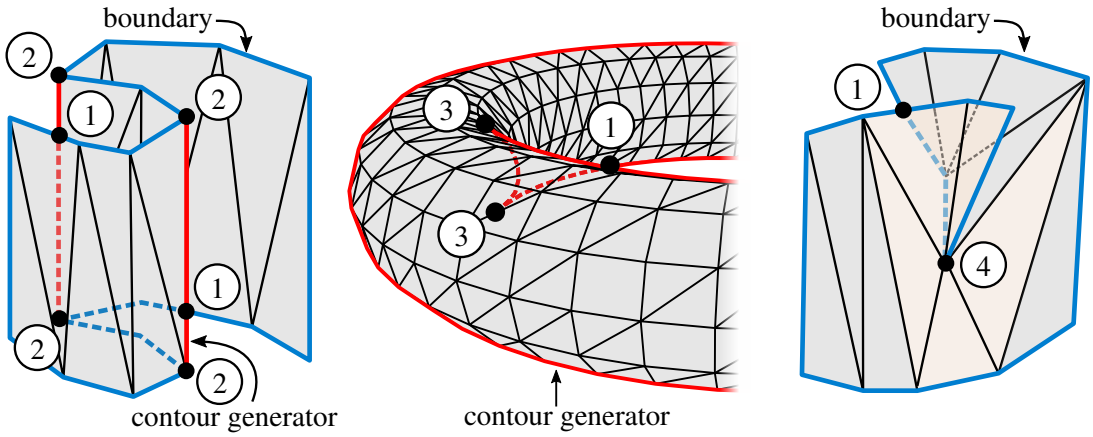


Figure 4.2: Singular points — From the camera viewpoint, (1) T-junctions at image-space intersections, (2) Y-junctions between a contour generator and two boundary edges, (3) contour generator curtain folds, (4) boundary curtain-folds. Contour generator edges are drawn in red, and boundaries in blue. Curtain folds are visualized in more detail in Figure 4.3.

4.3 Singular points

The contour curves are the set of contour edges, and the boundary curves are formed from the set of boundary edges (Figure 4.2). There are a few types of points on these curves where visibility may change. We call these points *singular points*, or *singularities*. Two points that are connected by a curve that does not pass through any singularities must have the same visibility. So we can perform a ray test at one curve point, and then propagate the result each direction along the curve until reaching singularities. This will drastically reduce the number of required ray tests. Singularities indicate places where visibility *might* change.

These singularity data structures are also be used to record the 2D topology of the set of curves, i.e., which curves connect to which, which will later be useful for stylization of the line drawing.

There are only a few different types of singular points:

(1) The visibility of a mesh curve may change at an **image-space intersection**. Specifically, when a contour or boundary edge overlaps another curve in image space (Figure 4.2 (1)), it indicates that part of the far curve is obscured by the surface closer to the camera. This splits the far curve into two segments, one of which must be invisible, and creates a T-junction between the near and far edges in image space. (The other segments may be invisible as well, if some other part of the surface occlude them.) Note, that while all intersections on the 3D surface are also 2D intersections, it is more robust to detect and handle them as a separate case, below.

(2) Curve visibility may change when two curves **intersect on the 3D surface**. For example, a contour generator may intersect a boundary curve (Figure 4.2 (2)). This

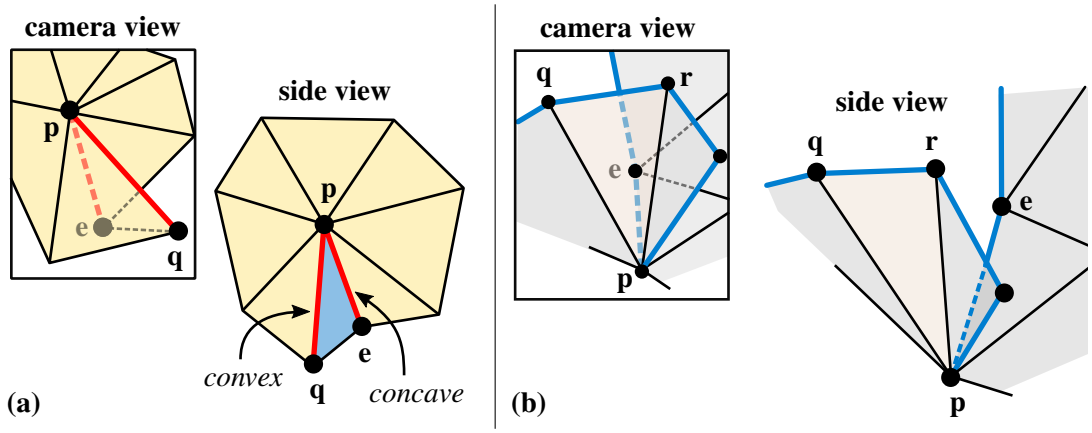


Figure 4.3: Curtain folds — A vertex **p** is a curtain fold if **(a)** it connects a *convex* contour generator edge to a *concave* edge, or **(b)** the edge **pe** is occluded by a face of the one-ring neighborhood of **p** (here, the triangle **pqr** in brown). (The former case **(a)** is a special case of the latter **(b)**.)

intersection can only occur at a mesh vertex, producing a Y-junction in image space (Grabli et al., 2010) if the three curve segments are visible, or may appear to form a continuous curve if one segment is hidden by the surface. In this case, the contour generator or the boundary curve may change visibility at the vertex. Surface intersection curves, on the other hand, lie within mesh faces, and thus intersect the contour generator within mesh edges.

(3) - (4) **Curtain folds** (Blinn, 1978) occur where the surface sharply folds back on itself in image space, causing the curve to become occluded by local geometry (Figure 4.3). More precisely, **a curtain fold occurs at a vertex connecting two curve edges, when one of the adjacent curve edges is occluded by another face connected to the vertex, and the other curve edge is not**. This definition can be used directly to identify boundary curtain folds. On contour generators, a simpler rule can be used: **a curtain fold occurs at any vertex where a convex contour edge meets a concave contour edge**, since concave contour edges must be invisible, and convex contour edges are locally visible. Details on detecting boundary curtain folds are given in Appendix C. Curtain folds are not important on other types of curves.

(5) A vertex may also connect more than two contour generator edges, in which case we call this vertex a **bifurcation** (Figure 4.4). In this case, there are no constraints on how the visibility can change; any adjacent edge of such a vertex can be visible or invisible. Boundaries can also exhibit bifurcations.

The above four cases are the only kinds of singular points for any surface curves. In implementation, different types of curves are handled separately, e.g., intersections on the surface need to be implemented with different cases for different kinds of intersecting curve, and intersections at vertices are handled separately from intersections within faces.

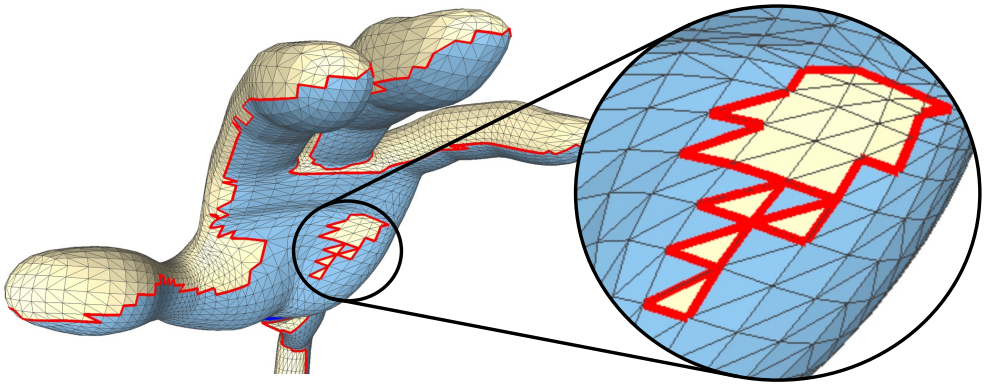


Figure 4.4: Bifurcations in the contour generator occur when more than two contour generator curves meet at a vertex. Closeup view on the mesh contours extracted from the smooth surface shown in Figure 1.4. "Red" ©Disney/Pixar

4.4 Visibility for other curve types

The above discussion is mainly for visibility-indicating curves: contours, boundaries, and surface-surface intersections. Computing visibility for other surface curves is generally simpler. The cases are the same: visibility may change when overlapped by a contour generator or boundary, or when intersecting a contour/surface-intersection on the surface. Curtain folds occur only for contours and boundaries, and not other curves. Convex/concave determination is only useful for contours and not other curves. Furthermore, ray tests for other curves are generally more numerically stable, if they are not themselves near contours.

Since we have assumed that back-faces are always invisible, any curve that lies within a back-face must also be invisible, as must an edge connected only to back-faces. For example, a boundary edge on a back-face is always invisible.

4.5 View Graph data structures

In order to propagate visibility, we will build a data structure called a View Graph (Figure 4.5). Later, this View Graph will be used to represent the image curves for stylization.

The View Graph stores the complete topology and geometry of the curves, in both 2D and 3D (Figure 4.6 and 4.7). It is composed of the line segments of each curve, and the singularities that connect them.

The View Graph is built implicitly, in the algorithm described in the next section. Each edge on the surface is converted to a *line segment* data structure. Each line segment stores both the 2D and 3D positions of its endpoints, and the curve type (e.g., contour, boundary, etc.). The segment stores a pointer to the mesh face or edge that it lies on; each endpoint stores the vertex or edge it came from, if appropriate.

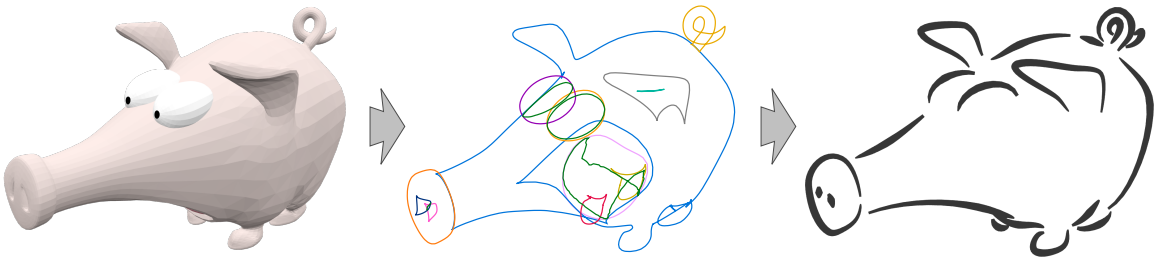


Figure 4.5: View Graph of Figure 1.1 — The View Graph structures the extracted contour curves by storing their topology and geometry. It serves as a support for both determining their visibility efficiently and generating stylized strokes.

Each line segment stores "head" and "tail" pointers, like a doubly-linked list. The "head" pointer points either to the next edge in the list, or else to a singularity object; likewise, the tail pointer points the other way. Each line segment also records whether or not it is visible; initially, visibility for all segments is marked as "unknown."

Each singularity records its type, and information specific to the singularity, e.g., a curtain fold points to the near and far line segments that it connects; an image-space intersection records the four line segments (near and far) that it connects.

Since any edge may have arbitrary numbers of overlapping curves, line segments may be broken repeatedly into smaller and smaller segments during construction of the View Graph.

As a simple example, once visibility is computed, one can draw a curve by starting at an arbitrary visible segment, and following pointers forward and backward, and continuing through singularities when possible, stopping only when the curve becomes invisible. Concatenating the 2D positions visited along the way yields a curve to draw.

4.6 Curve-based visibility algorithms

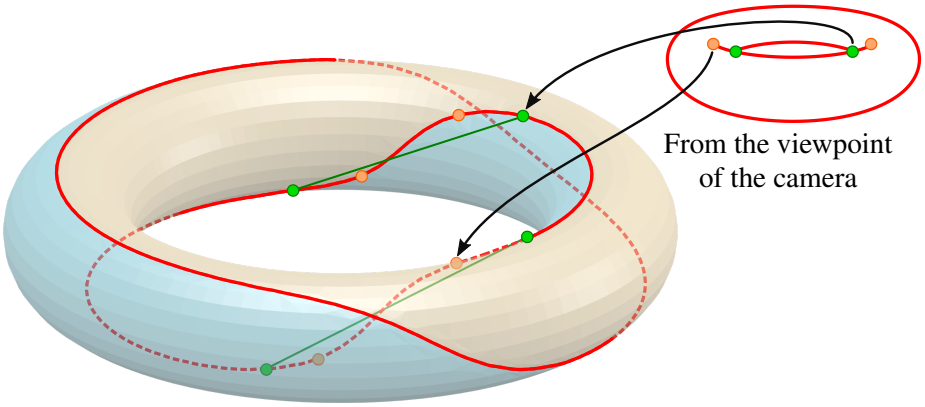
The basic visibility algorithm, is as follows:

1. **Detect all edges and project them to the image plane.** Each edge stores both the 2D and 3D coordinates of its vertices.

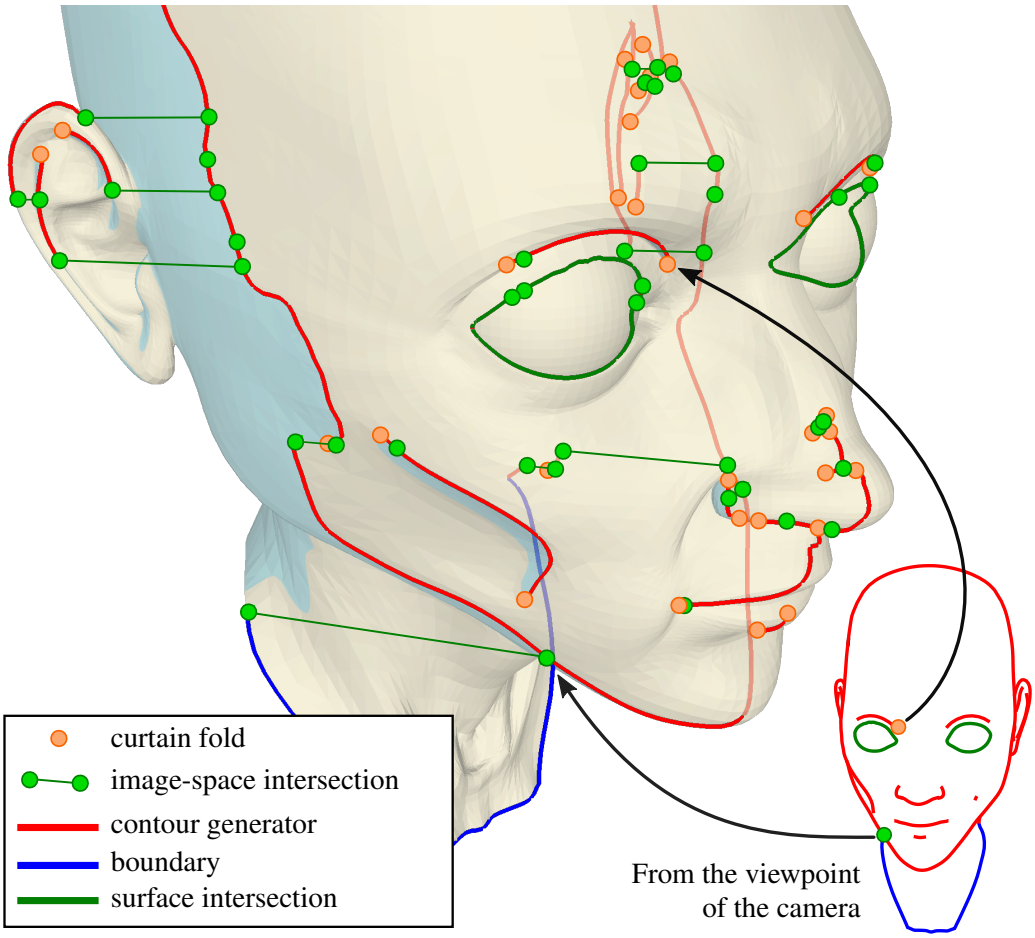
2. **Optionally, mark locally-invisible curves**: mark concave contour edges as invisible. Mark curves that lie entirely on, or adjacent to, back-faces as invisible.

3. **Insert a singularity at each curtain fold vertex**.

4. **Detect intersections on the surface**, i.e., when a boundary and contour edge pass through the same vertex, by iterating over all boundary vertices. Insert a singularity at the intersection point. Intersections involving two non-visibility-indicating curves can be ignored.

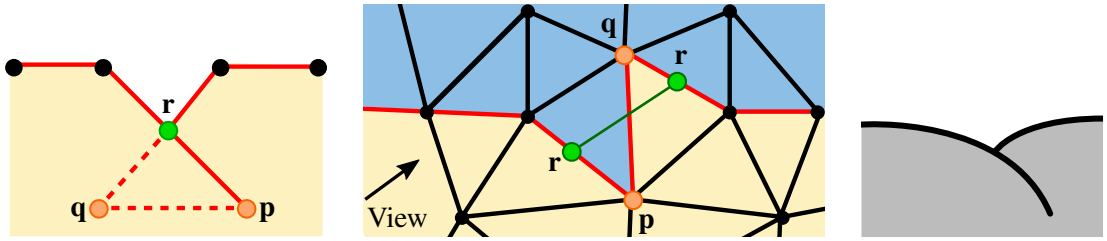


(a) View Graph of a torus



(b) View Graph of the "Angela" model © Chris Landreth

Figure 4.6: View Graph from (Bénard et al., 2014) — Line segments (contour generators, boundaries and surface intersections) are combined into chains that terminate at singular points (curtain folds, image-space intersections). This network of chains is called the View Graph. The graph on the right shows only the visible chains.



(a) Camera View **(b)** Overhead view **(c)** Rendering

Figure 4.7: The View Graph around a "fishtail", a common contour shape — Studying renderings like these, and how the 2D figure relates to the 3D drawing, is very helpful in understanding a specific rendering. **(a)** The "fishtail" shape in image space includes two cusps, and a partially-occluded contour. **(b)** The overhead view shows the contour's path over the surface. The contour separates front-facing and back-facing, and curtain fold cusps appear when the path switches direction. The curtain folds also separate convex from concave contours, which are always invisible. **(c)** A stylized rendering of this path produces this overlap drawing. This shape can occur at a large scale (e.g., a pair of hills or a puffy cloud), or at a subpixel level. In the latter case, we may wish to trim the extra bit of curve, as discussed in Section 9.3.

5. **Compute image-space intersections** between all pairs of edges; this can be done using a sweep-line algorithm in $O(n\log(n))$ time with n edges, e.g., (Bentley and Ottmann, 1979). Intersections where the near curve is not visibility-indicating can be ignored. Intersections on the surface should be ignored, since they are handled in the previous step. **Split the edges** at the intersection point and insert a singularity. The near edge does not need to be split since its visibility will not change at the intersection; doing so may still be useful for later stylization.

6. **For each edge where visibility is not yet marked, determine visibility** using a ray test to the center of the edge. Optionally, **propagate visibility** to adjacent edges, as described in the next section.

As a reminder, the visibility-indicating curves are contours, boundaries, and surface intersections. The above computations can be sped up by combining steps, e.g., the first four steps can all be performed with a single iteration over the mesh (or over the edges, for static meshes with one of the data structures of Section 3.7.2).

The above algorithm assumes that all curves lie on edges. For curves within edges, such as surface-intersections and hatching curves, the same basic procedure is used as well.

In the above computations, if invisible edges will never been drawn, then steps involving edges already known to be invisible can be skipped, to speed up computations. For example, concave contour edges can be omitted from the image-space intersection step. However, invisible edges are necessary for hidden-line rendering, and useful for QI propagation (described in the next Section); they are also very useful for visualization and debugging.

Markosian et al. (1997) also point out that curves adjacent to the scene's bounding box in image space must be visible. This is only useful for situations such as viewing only a single object in isolation, as opposed to entering a full 3D environment. A simple way to use this observation is to find the contour or boundary points with the maximum and mininimum x and y values; those points must be visible.

Visibility propagation. In the most basic version of the above algorithm, we perform a ray test for each edge that is not a concave contour. However, ray tests are computationally expensive, and we would like to perform as few of them as possible.

When two curve edges are connected at a shared endpoint, and there is no singularity, then the two edges must have the same visibility. Using this observation, we can propagate visibility after each ray test, following connections between edges until reaching a singularity. This simple propagation substantially reduces the number of ray tests required.

Implementation choices and numerics. In practice, there are many different ways to implement the algorithms in this section. One might first implement the vanilla ray-test algorithm. One then might implement grouping sequences of edges into singularity-free *chains*, and do one ray test per chain, then implement visibility propagation between chains. One can then add in additional constraints, e.g., concave contours must be invisible, and far edges of intersections must be invisible. At each phase of implementation, the algorithm should work correctly; each additional piece then accelerates the computation. On the other hand, implementing chains in the visibility pipeline adds considerable implementation complexity for a questionable amount of benefit.

There are multiple constraints on visibility that can be exploited for debugging. For example, if a ray test marks a concave contour as visible, then there is a bug or numerical error in either the concavity test or the ray test.

In practice, any of these tests can be corrupted by numerical errors. One heuristic is to ignore tests that are close to a threshold, or to vote among multiple tests (e.g., multiple ray tests at different points on a chain). Numerical issues are discussed more in Appendix C.

4.7 Quantitative Invisibility

We can propagate visibility information even further — and thus reduce the number of ray tests — by using the concept of *Quantitative Invisibility* (QI) (Appel, 1967; Markosian et al., 1997). The QI value of a point is the number of occluders of the point. A visible point has QI of zero (e.g., point $\mathbf{p}_2$ in Figure 4.1). A point blocked by two surfaces has QI of two. In practice, the QI of a point $\mathbf{p}$ can be computed by counting the number of mesh faces that intersect the line segment $\overline{\mathbf{pc}}$, excluding the face containing $\mathbf{p}$.

Ray tests can be expensive. Fortunately, the QI, or a bound on the QI, can be computed directly at certain surface points:

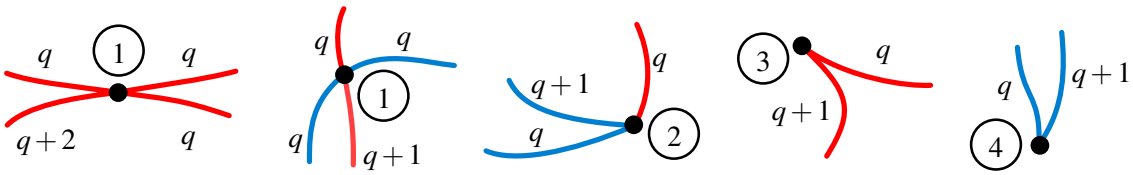


Figure 4.8: View Graph & QI propagation — QI values can be propagated at image-space intersections and curtain folds. (1) T-junctions at image-space intersections, (2) Y-junctions between a contour generator, two boundary edges, (3) contour generator curtain folds, (4) boundary curtain folds.

- **Front/back, convex/concave**: Points on back-faces, and concave contour edges, must have QI greater than zero.
- **The image-space bounding box of the scene must be visible**. For example, if a single object is viewed in isolation, all of the outer edges (the outer silhouette) must be visible. A practical test is to find all 2D edges with minimum and maximum x and y coordinates.

Once QI (or a bound) is computed at an edge, it may be propagated by the following rules:

- **Two edges that are connected on the surface without a singularity** must have the same QI.
- **Image-space intersection**: suppose that the nearest edge to the camera is a contour with QI value q. The occluded far edge must have QI of $q+2$, and the other side must have QI of q (Figure 4.8 (1)). If the occluding curve is a boundary, then the occluded far edge must have QI of $q+1$.
- **Intersection on the surface where a contour terminates at a boundary**. The boundary curve and the near contour generator must have the same QI of q. The far boundary edge may have QI of either q or $q+1$ (Figure 4.8 (2)). The specific value can be determined by a local overlap test, similar to the boundary curtain fold detection test.
- **At a curtain fold:** if the near edge has a QI of q, the far edge will have a QI of at least $q+1$ (Figure 4.8 (3),(4)). However, the far edge's QI could be higher in some exotic, unusual cases. For example, in a boundary curtain fold where the one-ring neighborhood spirals multiple times around the vertex like a fusilli pasta, the QI could increase more than 1. Hence, a local overlap test is necessary to count how many triangles in the vertex's one-ring overlap the far edge.
- **Other cases where multiple curves meet on the surface, and bifurcations**: the differences in QI between the adjacent curves can be determined using local overlap

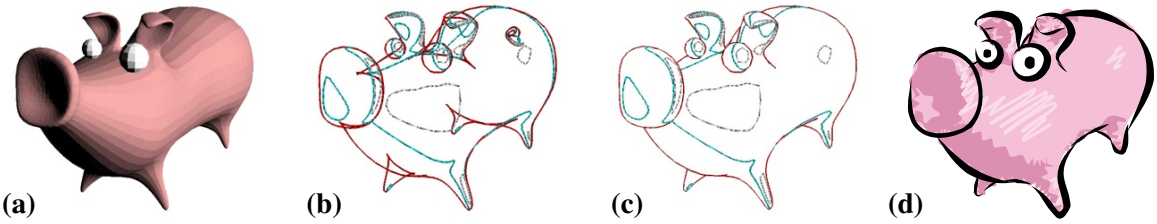


Figure 4.9: Planar Map (from Eisemann et al. (2008)) — Starting from a 3D model **(a)**, contours and isophotes are first extracted **(b)** and their visibility is computed by constructing a Planar Map **(c)** that yields a base vector depiction for stylization **(d)**. Isophotes are curves with constant shading, i.e., $(\mathbf{c}-\mathbf{p})\cdot\mathbf{n}=\alpha$ for some constant α.

tests, similar to the boundary curtain fold detection test. For example, if two edges meet at a bifurcation, and neither is occluded by any triangle in the one-ring, then they must have the same QI.

Hence, the resulting algorithm begins by first building the View Graph. The QI for most edges is initially marked as "unknown," though some edges can also be marked as "invisible" ($q>0$), such as concave contours. A single ray test is performed at some edge with unknown QI. By propagating this value through the view map, the QI can be determined for every edge in this edge's connected component. Hence, at most one ray test is necessary for each connected component. It is possible to determine some connected components' visibility without any ray tests at all. Propagating lower-bounds on QI increases the number of cases where this works. For example, a concave edge has $q>0$; if it is the near edge at a curtain fold, then the far edge has $q>1$.

4.8 Planar Maps

The Planar Map is a generalization of the View Graph that provides a more complete representation for artistic rendering: it represents not just the curves in a drawing, but also the regions between them. Given a Planar Map, one could theoretically stylize regions, strokes and their relationships in a more coherent way.

The Planar Map is a concept originally from graph theory. Given a set of 2D curves $\mathscr{C}$, the Planar Map corresponds to the *arrangement* $\mathscr{A}(\mathscr{C})$ of those curves, that is the partition of the plane into 0-, 1- and 2-dimensional cells (i.e., vertices, edges and faces) induced by the curves in $\mathscr{C}$. This partitioning is coupled with an incidence graph which allows navigation between adjacent cells.

Intuitively, the Planar Map is constructed by the following procedure. Specifically, all mesh faces are projected into the image plane. Faces that are completely occluded are discarded; faces that are partially occluded are subdivided in their visible and invisible parts, and their image-space adjacency information is updated.

If $\mathscr{C}$ contains the projection into the image plane of the contour generator and boundary curves, then the Planar Map $\mathscr{A}(\mathscr{C})$ corresponds to a generalization of the view graph presented in Section 4.5, one that includes the 2D regions bordered by the curves. By only keeping the closest cells to the camera, visibility can be determined (Figure 6.7).

Winkenbach and Salesin (1994) introduced the use of Planar Maps for stylized rendering. They used a 3D BSP tree to compute the visibility of the mesh faces (Fuchs et al., 1980), and a 2D BSP tree to build a partition of the image plane according to the visible faces. From this 2D BSP tree, they construct the Planar Map to have direct access to 2D adjacency information. They showed that this representation allows one not just to stylize contours, but to stylize the regions between the contours.

Computing visibility with BSP trees is both very expensive and numerically sensitive, even for simple models. A modern implementation of 2D arrangements is offered by the CGAL library (Fogel et al., 2012).

Eisemann et al. (2008) compute an approximate Planar Map by first computing the View Graph, and then joining regions between curves, identifying 3D correspondence using hardware buffers (Figure 4.9). This method is sufficient when a precise mapping between geometry and image space is not needed.

4.9 Non-orientable surfaces

As stated in Section 3.3, this tutorial assumes that all surfaces are orientable, and that only front-faces may be visible. It is also possible to generalize these algorithms to handle non-orientable surfaces (Figure 3.3), though with some additional complexity. This section outlines some of modifications to the algorithms of the last two chapters, though not all details will be spelled out.

We begin with contour detection between two triangles $\triangle\mathbf{abc}$ and $\triangle\mathbf{abd}$. We cannot directly use the front/back-facing test for contours, because facing direction is not defined on non-orientable surfaces.

For each triangle, there are two possible normals. For the first triangle, the possible normals are

$$\mathbf{n}_1 = \pm\frac{(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})}{||(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})||}.$$

The possible normals for the second triangle are similarly plus or minus the face normal.

In order to determine whether an edge is a contour, we must determine a locally-consistent pair of normals, that is $\hat{\mathbf{n}}_1$ which is either $\mathbf{n}_1$ or $-\mathbf{n}_1$, and $\hat{\mathbf{n}}_2$ which is either $\mathbf{n}_2$ or $-\mathbf{n}_2$. Consistent and inconsistent cases are visualized in Figure 4.10. The pair of normals is consistent as long as both normals agree as to whether the edge is convex or concave; see Appendix B.

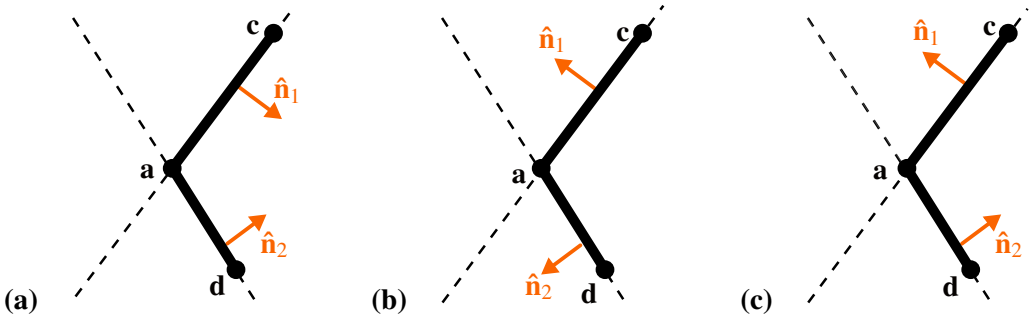


Figure 4.10: Valid/invalid configurations — Two adjacent triangles, shown in cross-section, with their assigned normals $\hat{\mathbf{n}}_1$ and $\hat{\mathbf{n}}_2$. The cross-section is some 3D plane perpendicular to the edge between the two triangles. **(a)** and **(b)** are valid configurations of the normals and **(c)** is invalid.

Once we have computed the locally-oriented normals $\hat{\mathbf{n}}_1$ and $\hat{\mathbf{n}}_2$, we can use the local sign test to determine if the edge is a contour, checking if the sign of $(\mathbf{a}-\mathbf{c})\cdot\hat{\mathbf{n}}_1$ is the same as the sign of $(\mathbf{a}-\mathbf{c})\cdot\hat{\mathbf{n}}_2$.

These local orientations cannot be reused when looking at other faces; when determining whether $\triangle\mathbf{abc}$ has a contour with one of its other neighbors, a local pair of normals must be computed for this pair of edges.

In general, any steps of the visibility algorithm that rely on the definition of front-facing or back-facing (or of convex or concave edges) must (a) be skipped if they are optional; (b) compute locally-consistent orientations before use; or (c) be replaced with a more general computation. For example, to determine if a vertex is a contour curtain fold, one could either compute locally-consistent orientations for the vertex's entire one-ring, or one could use the image-space self-overlap rule instead. One cannot assume that back-faces are invisible, as one can with oriented surfaces.

5 Fast Hardware-Based Extraction and Visibility

This chapter describes algorithms that use graphics hardware, such as multipass rendering and Graphics Processing Units (GPUs), to perform real-time contour detection and visibility. This improves performance over the CPU algorithms of the past few chapters, which can be very slow, especially for very complex geometry. Hardware-based methods can be very fast; in return, they do not guarantee correctness in all cases.

The earliest hardware methods directly produce visible contours using two rendering passes on the graphics card (Section 5.1), yet with limited stylization capabilities. Subsequent approaches massively parallelized the contour detection step (Section 5.2) or the visibility computation (Section 5.3) separately. They can be combined to render stylized lines at interactive framerates for complex 3D scenes.

5.1 Two-pass hardware rendering

The basic idea of these approaches is to render the geometry twice: first, to fill the depth buffer, and then second, using modified geometry, to make the contours emerge from the rasterization.

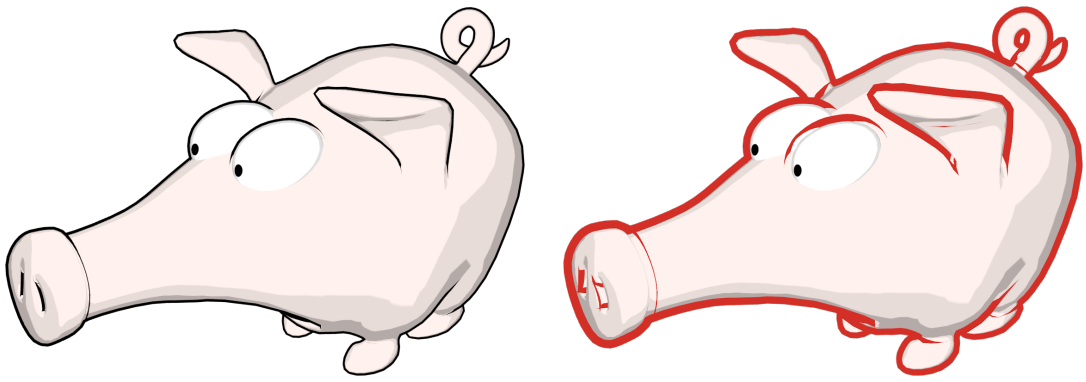


Figure 5.1: Two-pass hardware rendering — After a first standard rendering pass, the scene is enlarged and only back-faces are rendered; the visible parts of the back-faces produce the black edges (left). The back-faces scaling factor and color allow to control the line width and color (right). Image computed with Blender Solidify modifier (BlenderNPR, 2015b).

For instance, Rossignac and van Emmerik (1992) render the mesh in wireframe mode with thick lines, after a translation slightly away from the viewpoint. Alternatively, only back-facing polygons offset towards the camera can be rendered, such that they show through front-faces (Raskar and Cohen, 1999; Gooch et al., 1999). Using the first generation of programmable hardware, a similar effect is achievable in one pass by enlarging all back-facing polygons in the Vertex Shader (Raskar, 2001; Chen et al., 2015a).

Unlike the image-space filtering approaches (Chapter 2), two-pass methods provide more stylization options, namely, more control over line thickness and color (Figure 5.1). However, they do not provide stylization capabilities beyond these features. Furthermore, the need for two passes may make the method too slow for large models; nonetheless, they have been used in video games for simple models (St-Amour, 2010). Finally, these methods are especially useful for rendering contours of unstructured geometric representation such as point clouds (Xu et al., 2004).

Subsequent approaches independently accelerate the contour extraction or the visibility computation using the graphics card.

5.2 Contour extraction on the GPU

Once the graphics pipeline offered programmable stages with Vertex and Fragment Shaders, GPU implementations mirroring the brute force CPU algorithm described in Section 3.7 started to be possible. However, face adjacency information was not available initially. Card and Mitchell (2002); Brabec and Seidel (2003); McGuire and Hughes (2004) circumvented this limitation by drawing every edge of the mesh as a quadrilateral fin, storing as vertex attributes their two adjacent face normals. The dot product of these normals with the view direction can then be performed in the Vertex Shader, and non-contour vertices can be discarded.

With the introduction of the Geometry Shader stage, this is not required anymore (Stich et al., 2007). Regular mesh geometry with adjacency information can be sent to the GPU, and each face is then processed in parallel in the Geometry Shader. Some care must be taken not to detect the same contour edges twice, e.g., by discarding back-faces (Hermosilla and Vázquez, 2009). Sander et al. (2008) even obtained a speedup close to $2\times$ compared to this naive GPU implementation using a scheme for efficient traversal of mesh edges.

If the edges detected by the Geometry Shader are needed for a second rendering pass, a transform feedback operation can be used to read them back to the main GPU memory or even to the CPU.

5.3 Hardware-accelerated visibility computation

In this section, we present three hardware-based visibility techniques. In each case, the 3D curves are first detected either on the CPU or the GPU. The first two visibility techniques

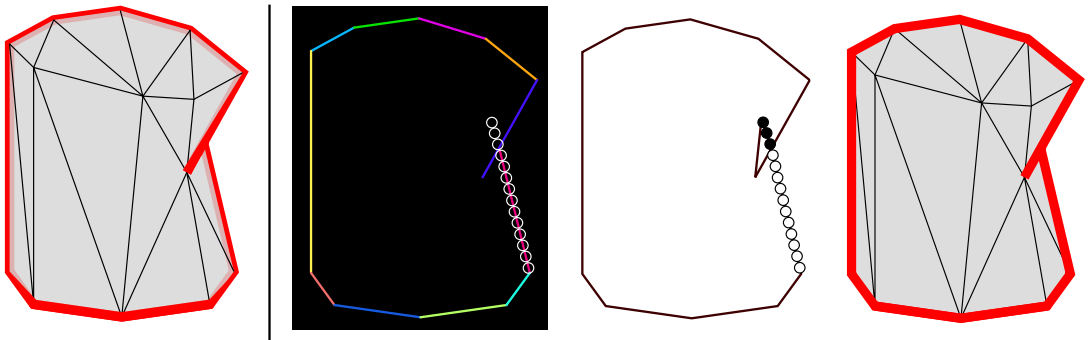


Figure 5.2: Buffer-based visibility — **(a)** Thick lines rendered with a simple depth test are irregularly occluded by the mesh geometry; **(b)** by first computing an item buffer with thin lines, and then probing visibility in this buffer (black and white circles) along the clipped lines, thick or stylized lines are correctly rendered. (The black wireframe is depicted for illustration purposes.)

work at image-space pixel precision and thus tend to suffer from aliasing artifacts, whereas the last method is mostly resolution independent. However, the first method is extremely simple to implement; in the simplest version, it just requires adding some 3D line segments in a 3D renderer.

5.3.1 Direct rendering with the depth buffer

Once the contour edges have been extracted either in software by one of the methods in Sections 3.7 and 6.2.1 or on the GPU with the previous technique, the simplest and fastest solution to determine their visibility is to use the standard depth buffer algorithm. First, the 3D scene is rendered with depth writes enabled — potentially disabling color writes if only lines should be rendered. The contour generator is then drawn as a 3D polyline with the less-or-equal depth function (e.g., with `glDepthFunc(GL_LEQUAL)` in OpenGL) and depth writes disabled. This way, line fragments occluded by the mesh geometry are automatically discarded during the depth test (Figure 5.2a). To avoid "depth fighting" between the polyline and the underlying surface, a small offset can be applied to the fragments' depth when rendering the mesh (e.g., with `glPolygonOffset` in OpenGL). Occlusion queries can be used if the result of the depth test needs to be read back on the CPU (Eisemann et al., 2008).

This technique works well for pixel-wide line rendering, but thicker lines may sometimes partially disappear in the geometry. It is difficult to stylize curves this way, other than using thick lines. In addition, this depth test is unstable because contour edges are often adjacent to faces that are almost parallel to the viewing direction, and thus both edges and faces project to the same depth buffer pixels, but the faces, rendered first, wrote depth values closer to the viewer. To address this issue, Isenberg et al. (2002) suggested modifying the depth test of a line fragment, by not only considering the single pixel of the depth buffer to which it

projects, but also its 8×8 neighborhood. Cole and Finkelstein (2010) proposed a full GPU implementation of such an approach, called the *spine test*, also suggesting computation of the depth buffer at a higher resolution to reduce artifacts due to undersampling. For detailed tutorial with a GLSL implementation of antialiased lines, see Rideout (2010).

5.4 Item buffer

An alternative solution proposed by Northrup and Markosian (2000) is based on an *item buffer*, which had previously been used to accelerate ray-tracing (Weghorst et al., 1984). The idea is to render each line into an off-screen buffer (e.g., a Framebuffer Object in OpenGL) with a thickness of one pixel and a unique color (ID). Each pixel of the item buffer eventually contains the unique color of a single visible line fragment at that pixel (Figure 5.2b). By scan-converting each line and reading the ID at the corresponding location in the item buffer, the visible portions of the line, called *segments*, can be determined. Each segment, or chain of segments, can then be rendered with thick lines, or even more complex stylization effects (Section 9.2). Kaplan (2007) showed that this approach can be extended to compute Quantitative Invisibility, thus allowing hidden line rendering with different styles, e.g., dotted lines. However, the item buffer suffers from two major limitations: it cannot be trivially anti-aliased, since line IDs cannot be averaged, and multiple lines cannot project to the same pixels even though they might all be partially visible. Cole and Finkelstein (2008) improved on those two aspects by computing a *partial visibility* for each line, using super-sampling and ID peeling, but with a significant memory overhead (typically $\times 12$ to 16).

5.5 Segment Atlas

Cole and Finkelstein (2010) circumvented the limitations of the item buffer by introducing a novel data-structure, called the *segment atlas*, that stores visibility samples along each line segment independently of their actual screen position. The segment atlas is created on the GPU in three steps (Figure 5.3). First, the input 3D lines are projected and clipped to the camera frustum with a dedicated fragment shader. For each 3D line $\overline{\mathbf{p}_i\mathbf{q}_i}$, the position of its endpoints $(\mathbf{p}'_i, \mathbf{q}'_i)$ in homogenous clip space are stored inside a GPU buffer along with a number l_i of visibility samples proportional to the screen space length of the line (potentially equal for maximum precision). During a second pass, a running sum turns the sample counts l_i into segment atlas offsets s_i. In a third step, the sample positions $\mathbf{v}_j$ are effectively created. Each clipped segment $\overline{\mathbf{p}'_i\mathbf{q}'_i}$ is discretized by generating a line from s_i to $s_i + l_i$ in a geometry shader and letting the rasterizer interpolate the endpoint positions. For each generated fragment, a shader performs the perspective division and viewport transformation to produce the screen-space coordinate $\mathbf{v}_j$ of the sample. The depth buffer is then probed at this position and the returned value is compared with the sample own depth value; the partial visibility resulting from this test α_j is written in the segment atlas, by construction, at the proper location. Finally, the line segments (or chains

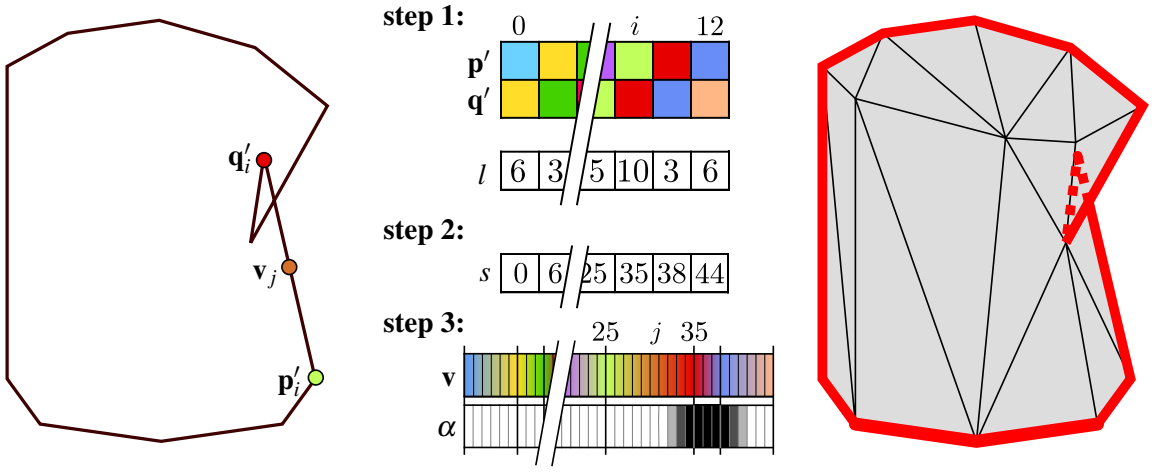


Figure 5.3: Segment atlas — Each input 3D line $\overline{\mathbf{p}_i\mathbf{q}_i}$ is projected and clipped by a fragment shader, which also computes its associated number of visibility samples l_i. These samples are then converted into segment atlas offsets s_i by a running sum. The positions $(\mathbf{p}'_i, \mathbf{q}'_i)$ are eventually interpolated and the resulting sample positions $\mathbf{v}_j$ are used to compute the partial visibility α_j at this location by reading the depth buffer. The clipped segments $\overline{\mathbf{p}'_i\mathbf{q}'_i}$ can then be rendered in screen-space leveraging partial visibility to modulate the style of the line. (The black wireframe is depicted for illustration purposes.)

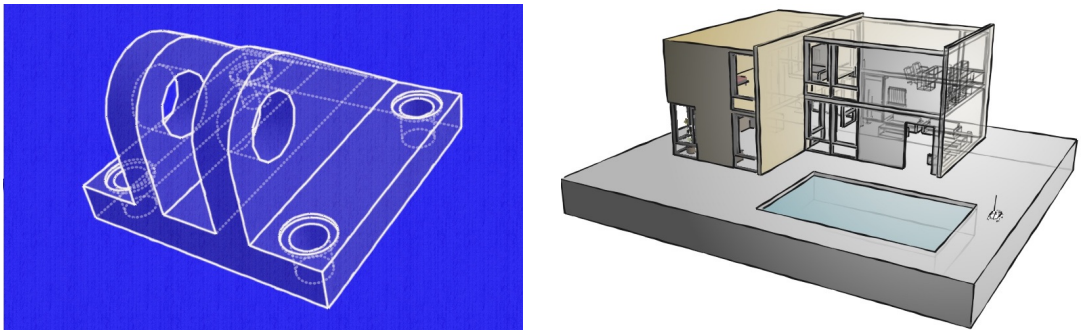


Figure 5.4: Stylized line renderings using the Segment Atlas algorithm (Cole and Finkelstein, 2010) — Hidden lines are included in these renderings. Images generated with "dpix" (Cole et al., 2010).

of segments with little modifications) can be rendered with arbitrary thickness and style using the fragment-level visibility information provided by the segment atlas (Figure 5.4). This method is up to 4× slower than direct OpenGL rendering (Section 5.3.1) for small 3D models, but 2× slower (or better) for complex meshes.

6 Smooth Surfaces as Meshes

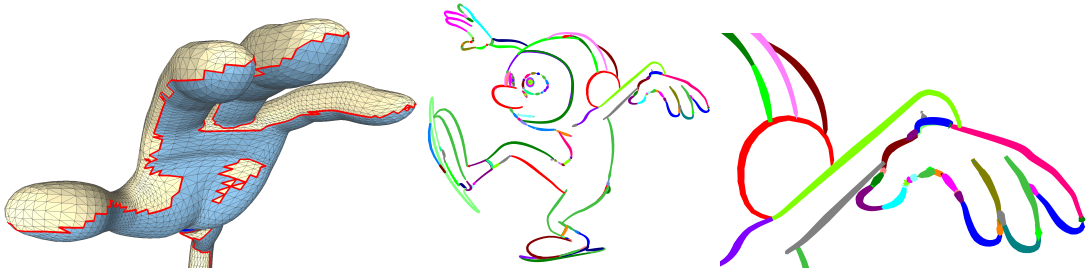
The algorithms described in the previous chapters work for polyhedral meshes. In this chapter, we describe heuristics for treating smooth surfaces as meshes for rendering. In Chapter 7, we will begin formal discussion of the theory and algorithms for smooth surfaces; using this theory avoids the problems with these heuristics.

6.1 The ups and downs of mesh rendering for smooth surfaces

Many previous researchers have taken the approach of converting their smooth surface into a triangle mesh, and then computing the contours of that mesh. This may seem like a sensible strategy, as it is common in computer graphics to tessellate a smooth surface and simply render the tessellation. If the contours will be directly rendered as line segments, e.g., thick black lines (as in Chapters 2), then this method produces good results.

Unfortunately, for stylized curves, this strategy leads to numerous artifacts, as illustrated in Figure 6.1. In fact, the topology of the contours will invariably get worse: smooth contours cannot exhibit bifurcations, but mesh contours can exhibit arbitrary branching.

Sometimes one starts from a mesh that begins polyhedral, such as geometry from web repositories or range scanners. Nonetheless, the contours may be unexpectedly messy,



(a) Mesh contours **(b)** Chained curves **(c)** Closeup view on **(b)**

Figure 6.1: Mesh contours of a smooth surface (Bénard et al., 2014) — The original surface is shown in Figure 1.4. **(a)** Converting the surface to a triangle mesh and extracting contours produces overly complex topology, including many bifurcations not present in the smooth surface's contours. **(b-c)** Connecting chains of edges that end at singularities produces many small chains, not directly suitable for stylization. Each chain is shown in a different color. "Red" ©Disney/Pixar

because the underlying surface is smooth, even if the representation is not. This may surprise the practitioner who is used to using triangle meshes for smooth objects.

Over the years, many researchers have observed the problems with mesh contours and developed heuristics to address them. Using these kinds of heuristics can be effective when speed and simplicity is valued over perfection. This chapter describes heuristics for identifying cleaner contours, and Section 9.3 describes heuristics for cleaning up mesh contours as a post-process.

Because these methods are heuristic, animating these curves usually produces some flickering artifacts, where curve sections appear and disappear. It is often argued that these artifacts are allowable, because hand-drawn animation typically exhibits some flickering that gives it a sense of imperfection and life. However, the flickering artifacts are often qualitatively different from hand-drawn animation: these artifacts are errors that humans would not normally make.

In our experience, every researcher who encounters these problem expects there to be a simple fix. They immediately propose their own clever solution. Efforts to turn these clever solutions into perfect results have always failed.

We caution the reader that these heuristics will never produce perfectly correct curve topology. If you convert a smooth surface to a mesh, then you have discarded information about the true contour topology, and cannot recover it from the mesh. Using these heuristics, one should expect that curves will occasionally be connected incorrectly, and outlines might temporarily vanish. If you are working in an environment where visual perfectionism is valued over expediency, then trying eliminate errors from mesh contour rendering will lead to endless frustration.

On the other hand, for most cases, these heuristics are usually good enough. Accurate visibility, as discussed in Chapter 7, is more difficult.

6.2 Interpolated Contours

The Interpolated Contours approach (Hertzmann and Zorin, 2000) produces smooth contours from meshes, by using some ideas from smooth surface contours, but applied on meshes. It produces cleaner contours by design, e.g., bifurcations are not possible. The approach is analogous to Phong shading in computer graphics, in which a mesh is treated as if it has smoothly-varying normals. Interpolated Contours has been used in many research projects, including Freestyle (Grabli et al., 2010), now in Blender.

The general outline of the method is the same as for mesh contours: detect contours, detect singularities, and then compute visibility by ray tests and visibility propagation. The main difference is that Interpolated Contours pass within faces rather than on mesh edges.

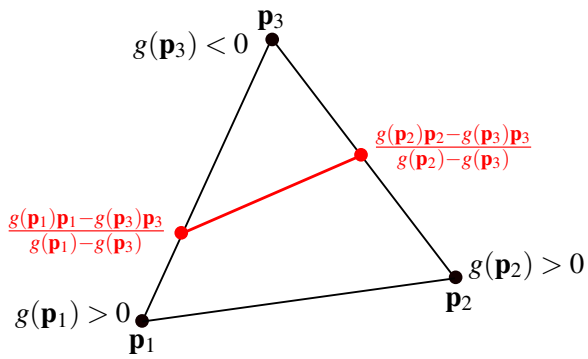


Figure 6.2: Linear interpolation — The function $g(\mathbf{p})$ is linearly interpolated along the oriented edges $(\mathbf{p}_3 - \mathbf{p}_1)$ and $(\mathbf{p}_2 - \mathbf{p}_3)$ of the triangle to find the endpoints of the line segment approximating the contour generator within the face.

6.2.1 Contour definition and detection

The approach is as follows. First, we assign a "fake" normal vector to each mesh vertex. As in Phong shading, this normal vector is a weighted average of the normals $\mathbf{n}_j$ of the adjacent faces, weighted by triangle areas A_j. This vector should be normalized to be a unit vector:

$$\hat{\mathbf{n}}_i = \frac{\sum_{j \in N(i)} A_j \mathbf{n}_j}{\sum_{j \in N(i)} A_j}$$
$$\mathbf{n}_i = \hat{\mathbf{n}}_i / ||\hat{\mathbf{n}}_i||$$

with $N(i)$ the one-ring face neighborhood of vertex i. One may also use the more robust weights of Max (1999).

For a given camera center $\mathbf{c}$, we define the orientation function g at a vertex i as:

$$g(\mathbf{p}_i) = (\mathbf{p}_i - \mathbf{c}) \cdot \mathbf{n}_i.$$

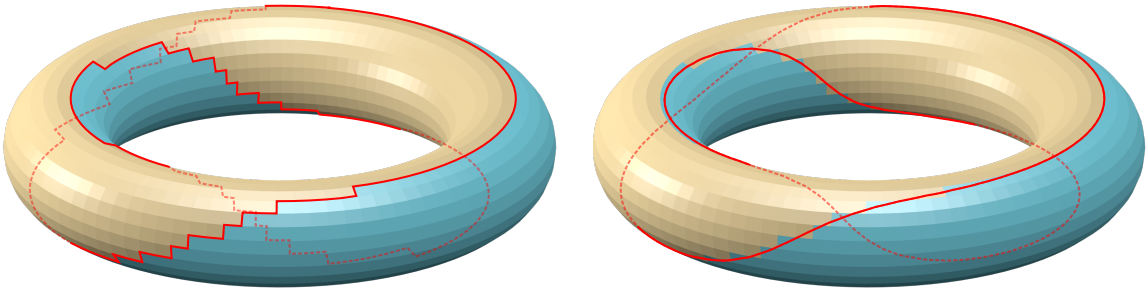
A vertex with $g(\mathbf{p}) > 0$ is considered to be front-facing, and with $g(\mathbf{p}) < 0$ is back-facing. The generic position assumption implies that we cannot have $g(\mathbf{p}) = 0$ at a vertex.

We then linearly interpolate $g(\mathbf{p})$ within the face, which is equivalent to linearly interpolating the normals within a face. This defines the orientation function over the entire surface as a piecewise linear function.

A face in which the orientation function has opposite signs at two vertices contains a contour. This contour is a line segment within the face (Figure 6.2). The endpoints of the line segment are found as follows.

On the edge between the vertex i and the vertex j, the linear interpolation is:

$$g(t) = (1-t)g(\mathbf{p}_i) + tg(\mathbf{p}_j).$$



(a) mesh contours **(b)** Interpolated Contours

Figure 6.3: Mesh vs. Interpolated Contours — Unlike mesh contours **(a)**, the piecewise linear approximation of the contour generator **(b)** crosses back-faces of the polygonal mesh and may thus be hidden by front-faces closer to the camera.

(a) Interpolated Contours **(b)** Chained curves **(c)** Closeup view on **(b)**

Figure 6.4: Interpolated Contours of a smooth surface (Bénard et al., 2014) — The original surface is shown in Figure 1.4. **(a)** Interpolated Contours are much smoother and have approximately correct topology (compared with Figure 6.1). **(b)** Chaining these curves gives much smoother, coherent curves. **(c)** Visibility is not well-defined for these curves, and gaps and other small errors may appear. "Red" © Disney/Pixar

A contour crosses this edge when the sign of $g(\mathbf{p}_i)$ is opposite the sign of $g(\mathbf{p}_j)$. Solving for $g(t) = 0$ gives the contour point position as (Figure 6.2):

$$t = \frac{g(\mathbf{p}_i)}{g(\mathbf{p}_i) - g(\mathbf{p}_j)},$$
$$\mathbf{p}(t) = (1-t)\mathbf{p}_i + t\mathbf{p}_j = \frac{g(\mathbf{p}_i)\mathbf{p}_i - g(\mathbf{p}_j)\mathbf{p}_j}{g(\mathbf{p}_i) - g(\mathbf{p}_j)}. \tag{6.1}$$

As illustrated in Figures 6.3 and 6.4, these contours typically have much smoother and coherent topology than the mesh contours.

6.2.2 Fast detection for static surfaces

The dual space data structures of Section 3.7.2 can be adapted to find Interpolated Contours on static meshes. For example, in the perspective dual space method of Hertzmann and Zorin (2000), each mesh vertex maps to a dual point $\mathbf{s} = (s_1, s_2, s_3, s_4) = (-n_x, -n_y, -n_z, \mathbf{n} \cdot \mathbf{p})$. As before, the camera maps to dual plane $g(\mathbf{s}) = (c_x, c_y, c_z, 1) \cdot s = 0$. A dual edge is drawn between each pair of adjacent vertices; a dual edge contains a contour point if the edge crosses the camera dual plane. Hence, finding all edges with contour points is again reduced to intersecting a plane with a set of line segments.

6.2.3 Singularities

The singularities of Interpolated Contours are similar to those of mesh contours. However, they behave somewhat differently. Interpolated Contours cannot exhibit bifurcations. Intersections on the surface lie within faces, since these contours lie within faces.

Defining curtain folds for Interpolated Contours requires the theory for smooth contours, which we will describe in the next chapter. For now, we will simply assume that we have a way to compute a function κ_r at each mesh vertex. This function, called the radial curvature, will be defined later in Section 7.4. Linearly interpolating this function across each face gives a function $\kappa_r(\mathbf{p})$ over the face, and a line segment with $\kappa_r(\mathbf{p}) = 0$ can be computed by linear interpolation across the face edges, just as was done for the contour generator. For a mesh face that contains zero crossings in both $g(\mathbf{p})$ and $\kappa_r(\mathbf{p})$, the curtain fold lies at the intersection of these two line segments, if they intersect (Hertzmann and Zorin, 2000; DeCarlo et al., 2003). For methods to compute curvature from meshes, see (Váša et al., 2016).

When detecting a curtain fold this way, there will often be a spurious image-space intersection between the contour and itself near the curtain fold, and one may need a heuristic to clean up this case. This can get tricky if other image-space intersections occur between these singularities.

6.2.4 Visibility

For ray tests, we use the original triangle mesh to determine when the smooth contour is occluded by the mesh (Hertzmann and Zorin, 2000). These computations are necessarily heuristic, because the Interpolated Contours are not the contours of the mesh. Some of the visibility techniques for mesh contours do not apply for Interpolated Contours; for example, it is unclear whether there is a useful analogue to "concave" and "convex" contours for Interpolated Contours, or whether QI can be propagated safely. As a result, the simplest choice is to use multiple ray tests per curve for visibility, whenever possible.

Unfortunately, the approximate contour generator is not the mesh contour generator. About half the segments of the approximate contour generator lie on back-faces of the triangle mesh (whatever the tessellation density of the mesh), and they are thus hidden by

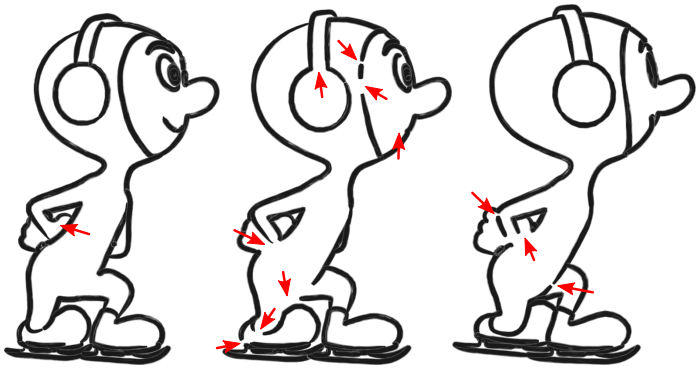


Figure 6.5: Stylized interpolated contours of a smooth surface (Bénard et al., 2014) — The original smooth surface is shown in Figure 1.4. After visibility computation, the contours exhibit many breaks and gaps (red arrows) which lead to objectionable temporal artifacts after stylization. "Red" ©Disney/Pixar

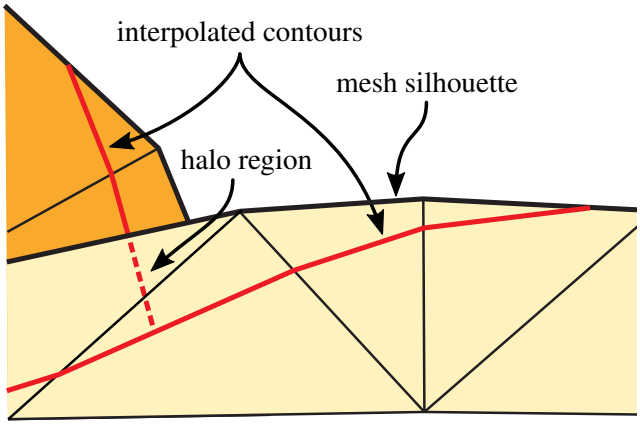
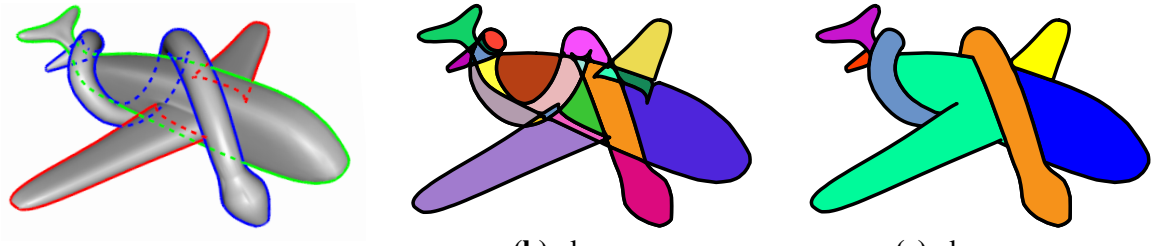


Figure 6.6: An example of the problem with interpolated contour visibility (Bénard et al., 2014) — In image space, the interpolated contour lies within the mesh contour. This creates a "halo region" between the two contours in which the surface occludes other surfaces but the mesh is invisible. Ray tests to the rear surface in this region will say the rear surface is invisible. In other configurations, such as nearly-flat, bumpy surface, a surface can "halo" other curves nearby on the same surface.

front-faces closer to the camera (Figure 6.3b). This leads to a number of visibility errors (Figure 6.5). Several heuristics have been proposed to mitigate this problem.

Hertzmann and Zorin (2000) use a voting scheme. Line segments between singularities are combined into chains, and then multiple ray tests are performed for each chain. These ray tests occur at the mesh vertices on faces with contours, using the vertices nearest to the viewer. The visibility is determined by a vote of these ray tests. In addition, Grabli et al. (2010) ignore occlusions from triangles adjacent to the contour's face. However these heuristics are not robust in every configuration.



(a) mesh contours

(b) planar map without visibility

(c) planar map with visibility

Figure 6.7: Planar Map — Starting from the projected mesh contours **(a)**, the Planar Map corresponds to the partition into cells of the 2D plane induced by these curves **(b)**. If only visible mesh faces are inserted in the Planar Map or snaxels are used, occluded contours will be discarded **(c).**

A more subtle issue is that the mesh silhouette does not line up with the smooth contour (Eisemann et al., 2008; Bénard et al., 2014). Ray tests are performed against the mesh, and the part of the mesh outside the interpolated contour can occlude other surfaces, making them incorrectly invisible (Figure 6.6). Other problem cases are discussed in Bénard et al. (2014).

6.3 Planar Maps

Planar Maps, as introduced in Section 4.8, represent all the visible strokes and regions within a graph. For mesh rendering from smooth surfaces, they offer the appeal that, even if there might be errors in the mesh approximation, the resulting drawing will still be internally consistent; it cannot have, e.g., giant holes in the outline.

Eisemann et al. (2009) presented a method to construct a Planar Map of the visible contours; their method involves a hybrid of mesh contours and Interpolated Contours. They first compute the view graph of the input scene, and then backproject it onto the mesh to define regions of constant visibility. Performing a ray test through each region center, they build an adjacency-occlusion graph by inserting links between successively intersected region pairs, as well as between adjacent regions. Contour edges at the frontier of a visible and an occluded region in this graph should be invisible in the Planar Map. The backprojection step is inspired by 3D BSP tree construction and involves many plane-triangle intersections that may thus suffer from the same numerical issues, requiring special tolerancing.

Karsch and Hart (2011) avoided these robustness complications by computing jointly the partition on the mesh and in the image plane. They use *snaxels*, i.e., active contours (Kass et al., 1988) whose vertices lie on mesh edges, to delineate the zero sets of an implicit contour function defined over the mesh surface. While seeking to minimize the energy functional, the snaxels obey topology rules for splitting and merging. By detecting snaxels collisions both on the 3D surface and in the image plane, and designing proper merging rules, Karsch and Hart (2011) ensure that no contours are intersecting if they belong to separated

regions of the mesh (Figure 6.7). The main problem of this approach is its initialization: one snaxel front per Planar Map region should be seeded on the 3D geometry, but these regions are not known a priori. Multiple passes, introducing additional fronts, might thus be required to converge to the correct solution.

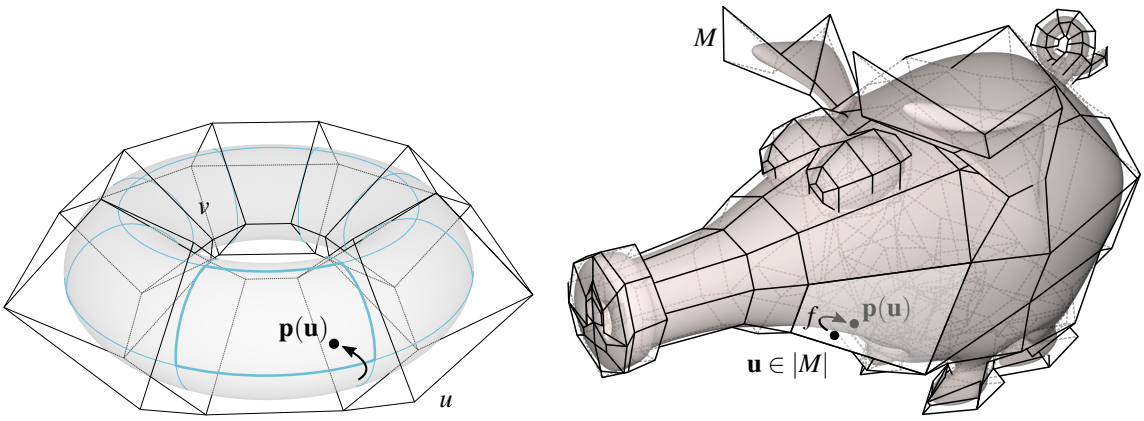
7

PARAMETRIC SURFACES: CONTOURS AND VISIBILITY

In the previous chapters, we only considered polygonal meshes as input. We will now present the theory of contours on smooth surfaces. Most of this theory mirrors that of mesh contours, but using the tools of differential geometry, whose fundamentals are briefly summarized in Appendix A. This chapter will focus on parametric surfaces; implicit surfaces and volumes will be described in Chapter 8. Algorithms to extract the apparent contour of smooth surfaces yield mostly-correct results for most surfaces. However, as we discuss, perfectly computing smooth surface contours remains an open research problem.

7.1 Surface definition

In the following, we will assume that all surfaces are at least C^1 smooth everywhere, though it is conceptually straightforward to generalize to surfaces with creases, since they behave like mesh edges (Chapter 3). The theory in this chapter applies to any surface with a parametrization $\mathbf{u}$, and a surface function with position $\mathbf{p}(\mathbf{u})$ and normals $\mathbf{n}(\mathbf{u})$, but the algorithms are designed for spline patches and subdivision surfaces. We briefly review these surfaces.



(a) NURBS surface (32 patches) **(b)** Subdivision surface (381 faces)

Figure 7.1: Parametric surfaces — The map f from the parameter plane $[0,1]^2$ or the control mesh surface $|M|$ to $\mathbb{R}^3$ defines the surface of a NURBS patch **(a)** or subdivision surface **(b)** respectively.

Spline patches. Spline patches (also called "freeform surfaces") are parametric functions from a 2D domain to a surface in 3D: $f : \mathbb{R}^2 \to \mathbb{R}^3$. Specifically, each input coordinate $\mathbf{u} = (u, v)$ maps to a 3D point:

$$\mathbf{p}(\mathbf{u}) = \begin{bmatrix} f_x(u,v) \\ f_y(u,v) \\ f_z(u,v) \end{bmatrix}$$

on the surface. In a spline patch, these functions are defined as linear combinations of basis functions applied to control points. For example, in the nonuniform rational B-spline (NURBS) patch. The shape of such a patch is parameterized by a grid of $(m+1) \times (n+1)$ control points $\mathbf{p}_{i,j} \in \mathbb{R}^3$ and their associated scalar weights $w_{i,j} \in \mathbb{R}$. The 3D surface is then given by:

$$\mathbf{p}(u,v) = \frac{\sum_{i=0}^{n} \sum_{i=0}^{m} N_i^k(u) N_j^l(v) w_{i,j} \mathbf{p}_{i,j}}{\sum_{i=0}^{n} \sum_{i=0}^{m} N_i^k(u) N_j^l(v) w_{i,j}}, \tag{7.1}$$

where N_i^k is the B-spline basis function of degree k for the i^{th} control point (Figure 7.1a). Details can be found in most computer graphics textbooks.

Regardless of the specific type of surface used, the surface normal at a point $\mathbf{p}$ can be computed as follows. The two 3D vectors:

$$\mathbf{t}_u(\mathbf{u}) = \left.\frac{\partial \mathbf{p}}{\partial u}\right|_{\mathbf{u}} = \left[\frac{\partial f_x}{\partial u}, \frac{\partial f_y}{\partial u}, \frac{\partial f_z}{\partial u}\right]^\top \quad \mathbf{t}_v(\mathbf{u}) = \left.\frac{\partial \mathbf{p}}{\partial v}\right|_{\mathbf{u}} = \left[\frac{\partial f_x}{\partial v}, \frac{\partial f_y}{\partial v}, \frac{\partial f_z}{\partial v}\right]^\top$$

are tangent vectors at $\mathbf{p}$. A surface normal at that point is:

$$\mathbf{n}(\mathbf{u}) = \mathbf{t}_u(\mathbf{u}) \times \mathbf{t}_v(\mathbf{u}),$$

Subdivision surfaces. Modeling surfaces of general topology is quite difficult with patches. Subdivision surfaces are a generalization of splines that are popular for modeling surfaces of arbitrary topology (Zorin and Schröder, 2000).

A subdivision surface is defined by a polygonal mesh and a refinement scheme. The input polygonal mesh is called the control mesh. The corresponding smooth surface, called the *limit surface*, is defined from the control mesh by recursively applying the refinement scheme an infinite number of times.

The topology of the control mesh, denoted M, provides a piecewise parameterization of the limit surface. (Since the control mesh must be a simple polyhedron, it may be deformed or even lifted to $\mathbb{R}^4$ to remove all self-intersections.) In particular, let $\mathbf{u} \in M$ be a point on the control mesh. Then the subdivision surface may be viewed as a function $\mathbf{p}(\mathbf{u}) : M \to \mathbb{R}^3$, defined by the subdivision scheme and the positions of the control vertices. The point $\mathbf{u}$ is called the *preimage* of a point $\mathbf{p}(\mathbf{u})$ on the surface. Analytic representations of $\mathbf{p}(\mathbf{u})$ and its normals $\mathbf{n}(\mathbf{u})$ have been derived for popular schemes, such as Loop (Loop, 1987; Stam, 1998b) and Catmull-Clark (Catmull and Clark, 1978; Stam, 1998a). The open source library "OpenSubdiv" (Nießner et al., 2012; Pixar, 2019) supports direct evaluation of limit positions, normals and curvatures for both Loop and Catmull-Clark surfaces.

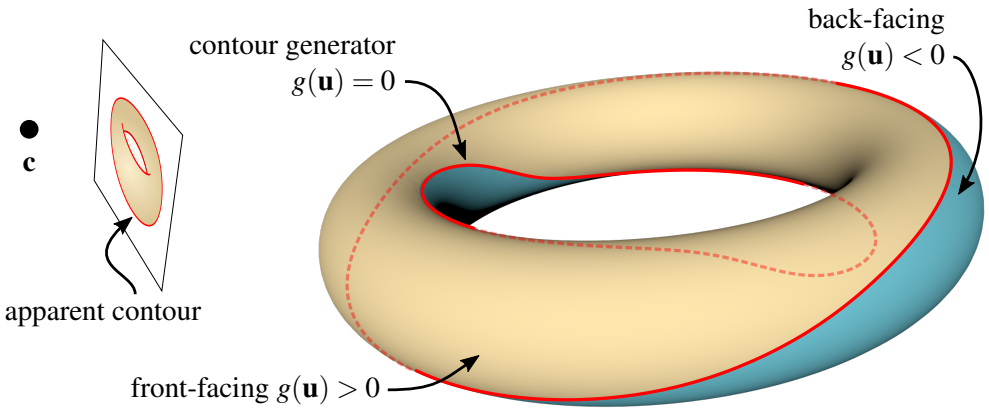


Figure 7.2: Smooth surface contour — The contour generator is the zero-set of the implicit orientation function $g(\mathbf{u})$ and thus the boundary between the front-facing and back-facing parts of a surface, as seen from a camera center $\mathbf{c}$. The apparent contour is the visible projection of the contour generator onto the image plane.

7.2 Contour definition

As before, we assume that the surface is oriented, in generic position, and that only front-facing points may be visible. The surface is viewed from a camera center $\mathbf{c}$. We define the *orientation function* (Figure 7.2):

$$g(\mathbf{u}) = (\mathbf{p}(\mathbf{u}) - \mathbf{c}) \cdot \mathbf{n}(\mathbf{u}).$$

A point with $g(\mathbf{u}) > 0$ is front-facing and a point with $g(\mathbf{u}) < 0$ is back-facing, the contour is the boundary between these regions: $g(\mathbf{u}) = 0$. More formally, following Definition 3.4.1. The contour is defined by:

Definition 7.2.1 (parametric contour generator)**.** *The collection of all points* $\mathbf{p}(\mathbf{u})$ *for which the preimages* $\mathbf{u}$ *satisfy* $g(\mathbf{u}) = 0$ *is called the* contour generator *(Marr, 1977). The visible projection of the contour generator onto the image plane is called the* apparent contour*, or, simply,* contour.

Interestingly, the smooth contour can also be interpreted in terms of shading, in two different ways. In the first way, we imagine photorealistic rendering of the surface with Lambertian shading ($\mathbf{n} \cdot \mathbf{v}$), with white texture, against a white background. The black pixels of this rendering ($\mathbf{n} \cdot \mathbf{v} \approx 0$) are the contour (Figure 7.3). Generalizing this idea of finding the darkest pixels (not necessarily black) of the Lambertian image motivates contour generalizations like the Suggestive Contours (DeCarlo et al., 2003; Lee et al., 2007) and isophote stroke thickness (Goodwin et al., 2007). And, an inverse interpretation is that the contours appear to have the same shape as rim lighting, in which an object is illuminated by a ring of lights perpendicular to the camera direction (Figure 7.4(a)).

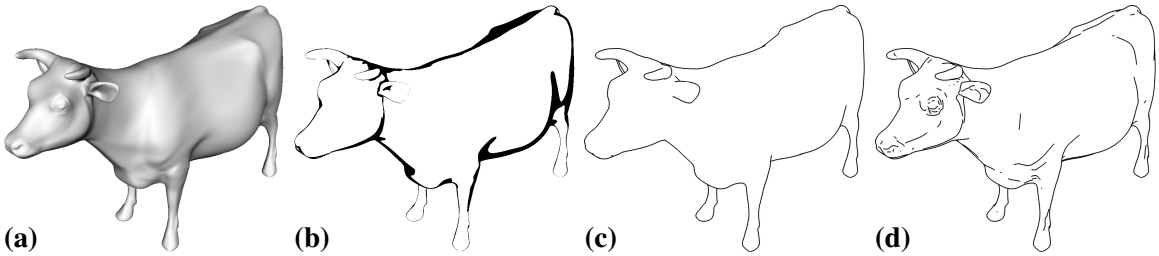


Figure 7.3: Shading interpretation of contours — **(a)** Lambertian shaded white object with light at viewpoint, so that shading is $\mathbf{n} \cdot \mathbf{v}$. **(b)** Thresholded rendering, for visualization **(c)** The contours are the black points in the shading image, where $\mathbf{n} \cdot \mathbf{v} = 0$. **(d)** Identifying dark ridges in the shading image produces the contours and Suggestive Contours (DeCarlo et al., 2003; Lee et al., 2007). Images generated with "qrtsc" (Cole et al., 2011).

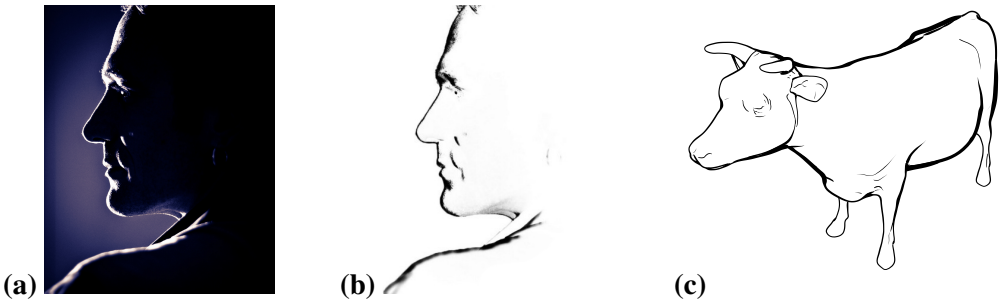


Figure 7.4: Rim lighting and rendering — **(a)** Photograph taken with rim lighting, i.e., a ring of lights perpendicular to the camera direction. Rim lighting approximates the occluding contour. (Photo by Flickr user japrea ⓒⓘⓢ) **(b)** Rim light photograph inverted and converted to grayscale, with the background removed. **(c)** Computer-generated line drawing using contours, Suggestive Contours (DeCarlo et al., 2003) and isophote thickness (Goodwin et al., 2007). The stroke thickness varies in the same way as it would for contours produced by rim lighting.

7.3 Contour extraction

Because the contour generator is an implicit polynomial function, we cannot directly compute it. Instead, we must numerically approximate it; existing methods approximate it by piecewise linear curves. While an early method proposed marching along the contour in parameter space (Hornung et al., 1985), more modern methods identify patches with sign changes of $g(\mathbf{u})$ (Elber and Cohen, 1990; Gooch, 1998; Bénard et al., 2014), similar to the treatment of Interpolated Contours in Chapter 6.

Specifically, we first evaluate the orientation function $g(\mathbf{u})$ at all control vertices (Figure 7.5). For a spline patch, these control vertices can be visualized on a regular grid; for a subdivision surface, they live on the control polygon. We denote points with $g(\mathbf{u}) > 0$ as F and $g(\mathbf{u}) < 0$ as B.

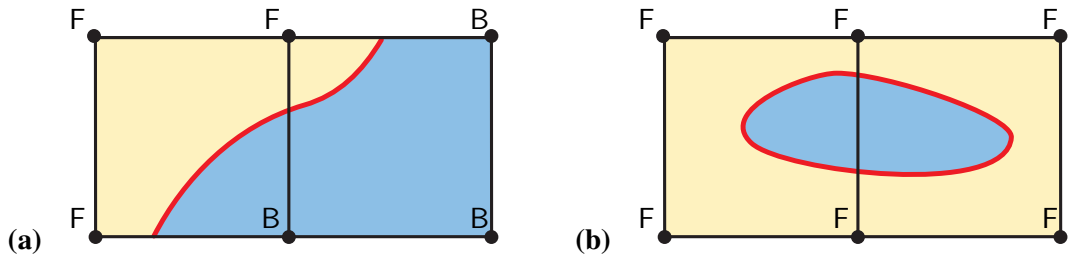


Figure 7.5: Orientation function evaluated at the control vertices — When $g(\mathbf{u})$ has opposite signs (F and B) on both ends of a control polygon edge **(a)**, a contour point must exist on that edge. When there is no sign change (F and F or B or B), there may be zero contour points, or a larger even number of contour points per edge **(b)**.

For any edge on the control polygon with opposite signs on the edge (F and B), there must be a contour point somewhere on that edge (Figure 7.5a). For edges with the same sign (F and F or B or B), there *might* be contour points, in some even number, such as a small loop centered on this edge (Figure 7.5b). In the following algorithms, we generally assume that no small loops like this occur, and assume that no sign change indicates that there is no contour on the edge. (For the special case of rational splines under orthographic projection, Elber and Cohen (1990) showed a sign test that may be used to quickly identify that some patches cannot contain contours.)

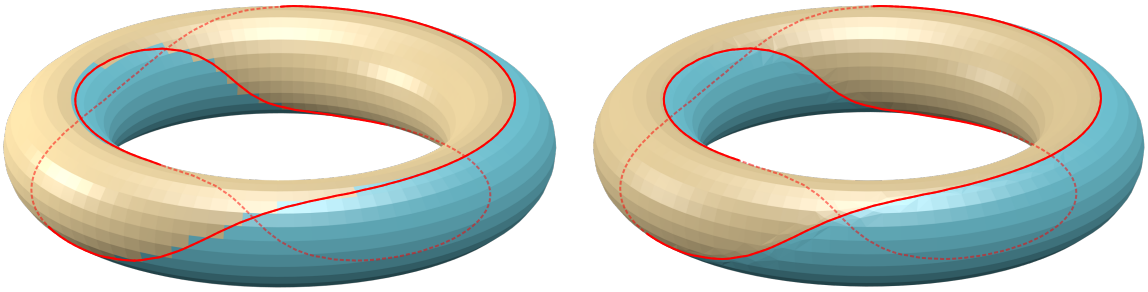
For each edge that must contain a contour, the edge can be parameterized as a 1D function $\mathbf{u}(t) = (1-t)\mathbf{u}_0 + t(\mathbf{u}_1)$ where $\mathbf{u}_0$ and $\mathbf{u}_1$ are the preimages of the control points. The preimage and position of the contour may be found using a root-finding algorithm on $g(\mathbf{u}(t)) = 0$ (Elber and Cohen, 1990; Bénard et al., 2014), such as the secant method or bisection search. A simpler approach is to linearly interpolate to approximate the contour location (Gooch, 1998), similar to Equation 6.1.

The identified contour locations may be connected to produce a piecewise linear approximation to the contour (Figure 7.6). A more precise curve may be found by subdividing control faces and repeating the root-finding process.

7.4 Contour curvature

We now discuss the curvature of 2D contours and 3D contour generators. This analysis is necessary to identify curtain folds, and also gives insight into the relationship between surface curvature and apparent contour curvature.

In order to analyze image contours, it is useful to consider the following tangent direction at contour point $\mathbf{p}$. The direction $\mathbf{w}$ is defined as the (unnormalized) projection of the view vector $\mathbf{v} = \mathbf{p} - \mathbf{c}$ onto the tangent plane at $\mathbf{p}$. For contour points, $\mathbf{w} = \mathbf{v}$ since $\mathbf{v}$ is already in the tangent plane. The normal curvature along $\mathbf{w}$ is called the *radial curvature* $\kappa_r(\mathbf{p})$ (Figure 7.7a) (DeCarlo et al., 2003; Koenderink, 1984). (See Appendix A.2 for the definition of normal curvature.)



(a) interpolated contours **(b)** mesh contours after consistent tessellation

Figure 7.6: Contour generator approximation — An input smooth torus represented as a Catmull-Clark subdivision surface is uniformly tessellated with one round of subdivision. With contour-consistent tessellation **(b)**, the mesh contours of the polygonal mesh is both topologically equivalent to the smooth surface contour and at the boundary of visible and invisible. Interpolated contours **(a)** do not have this property, leading to problems with visibility.

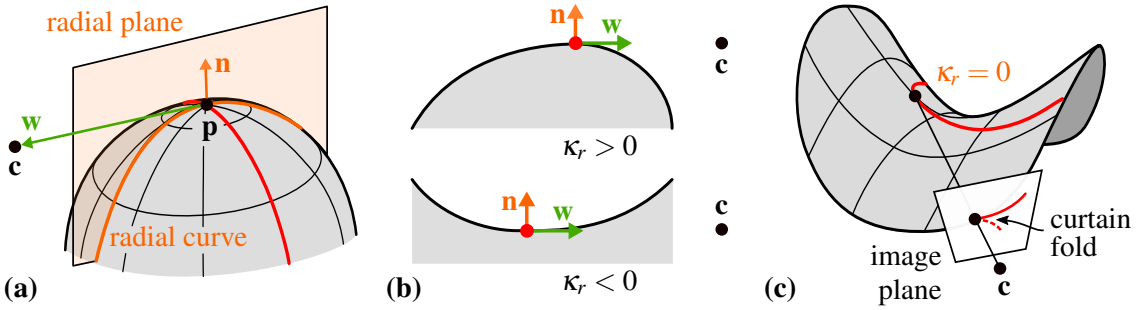


Figure 7.7: Radial curvature — **(a)** The radial curvature $\kappa_r(\mathbf{p})$ is the curvature of the radial curve at $\mathbf{p}$; **(b)** κ_r is necessarily positive for visible contours (top) otherwise it would be locally occluded by the surface (bottom), and zero at curtain folds **(c)**.

Another way to state the definition is as follows. The radial curvature is based on the *radial plane*, the plane that contains the point $\mathbf{p}$, the surface normal $\mathbf{n}$, and the view vector $\mathbf{v}$. The *radial curve* is the intersection of the surface with the radial plane. The radial curvature $\kappa_r(\mathbf{p})$ is then defined as the curvature of the radial curve at $\mathbf{p}$.

A contour can only be visible when it has a positive radial curvature ($\kappa_r > 0$) — otherwise the contour generator would locally lie in a surface concavity (Figure 7.7(b)). This exactly parallels the concepts of *concave* and *convex* contours on meshes: a contour with positive radial curvature is a convex contour (in the radial direction).

The *apparent curvature* $\kappa_p(\mathbf{p})$ of the contour curve is the curvature of the apparent contour at $\mathbf{p}$. Under perspective projection, Koenderink (1984) demonstrated that the Gaussian curvature K of the surface at $\mathbf{p}$ is related to the radial and apparent curvatures by:

$$K = \frac{\kappa_r(\mathbf{p})\kappa_p(\mathbf{p})}{\|\mathbf{p}-\mathbf{c}\|}.$$

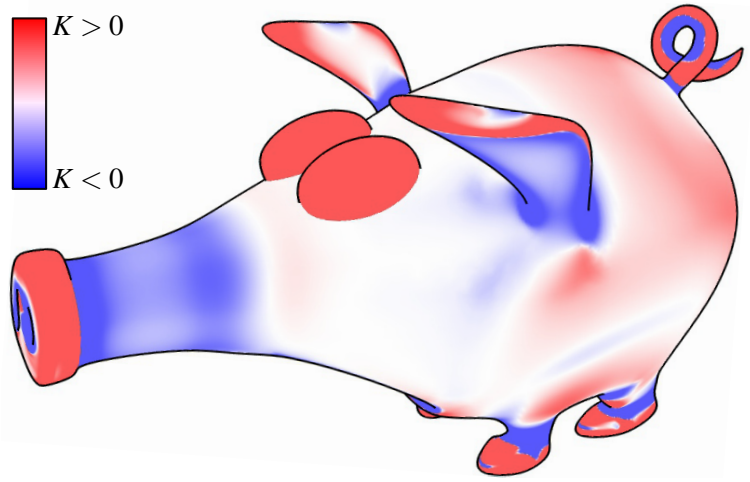


Figure 7.8: Relationship between the surface Gaussian curvature and the contour apparent curvature — Concave apparent contours originate from hyperbolic regions (in blue) and convex ones from elliptic regions (in red); their inflection coincides with parabolic points on the surface. Image generated with "qrtsc" (Cole et al., 2011).

The above observations allow us to relate the image-space curvature of the 2D contour with the corresponding 3D region. For visible contours, the sign of the apparent curvature is thus the same as the sign of the Gaussian curvature. If the surface is elliptical ($K > 0$), the fact that visible contours cannot have $\kappa_r < 0$, implies that κ_p is necessarily positive, and thus the apparent contour displays a convexity. Conversely, if the surface is hyperbolic ($K < 0$), $\kappa_p < 0$ and thus the apparent contour is concave. This leads to the following general rule illustrated in Figure 7.8: a convex apparent contour corresponds to a convex surface, a concave contour implies a saddle-shaped surface, and an inflection on the contour ($\kappa_p = 0$) implies a parabolic point on the surface ($K = 0$).

7.5 Singular points

Curves on smooth surfaces exhibit similar *singular points* — that is, points where visibility might change — as on meshes. Image-space intersections, intersections on the surface, and curtain folds are all possible singularities. However, smooth surface contours may not exhibit bifurcations in generic position (Hertzmann and Zorin, 2000).

Intersections. Image-space intersections and intersections on the surface create singularities for curves on smooth surfaces.

Finding intersections on the surface between boundaries and other curves typically involve root-finding along each boundary edge, e.g., for boundary-contour intersection, find the boundary edge point with $g(\mathbf{p}(t)) = 0$.

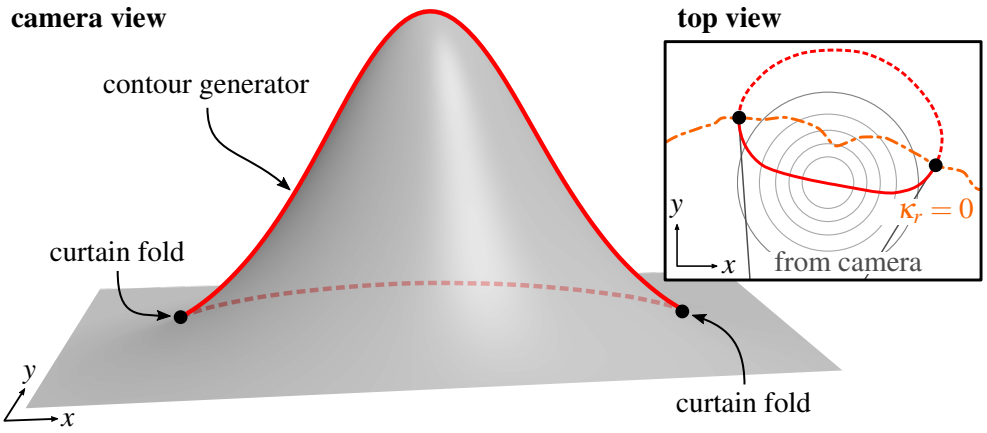


Figure 7.9: Contour generator curtain folds on a smooth surface — As shown on the top view, curtain folds are at the intersection of the contour generator and radial curvature zero-isocurve. At each curtain fold, the curve tangent is aligned with the view direction.

Image-space intersections involving contours are more difficult to find, because contours are implicitly defined. These intersections must be detected numerically, and there is no simple data-structure for accurately accelerating the search without the possibility of missing some intersections. There are two general strategies one can take. First, one can convert the contours into polylines, and compute the intersections of these polylines. This is simple but may often be incorrect in some cases. Second, one may use an adaptive subdivision approach, in which bounding boxes for each curve are subdivided until either an intersection is found or the absence of an intersection can be proven (Elber and Cohen, 1990).

Computing intersections between smooth surfaces is also difficult (e.g., (Houghton et al., 1985)), and these intersection curves must then be intersected with other curves on the surface.

Contour curtain fold definition. Koenderink (1984) demonstrated that the radial curvature vanishes at a *curtain fold cusp* ($\kappa_r = 0$), the contour transitioning from invisible to potentially visible. At a curtain fold, the 3D tangent of the contour generator exactly coincides with the view vector. As a result, the projection is not smooth (κ_p is infinite); hence, curtain folds correspond to cusps in the apparent contour (Figures 7.7c and 7.9). It can be shown that these points are the only generic cusps of smooth surface contours. For this reason, many previous authors use the term cusp instead of curtain fold; we use the latter terminology to emphasize the correspondence with curtain folds on mesh contours.

As with meshes, curtain folds occur at the transition from concave ($\kappa_r < 0$) to convex ($\kappa_r > 0$) contours: when the contour transitions from locally occluded (concavity) to locally visible (convexity). In the vicinity of curtain folds, the surface is necessarily hyperbolic; the visible branch of the apparent contours must thus be concave in image space (Koenderink and van Doorn, 1982).

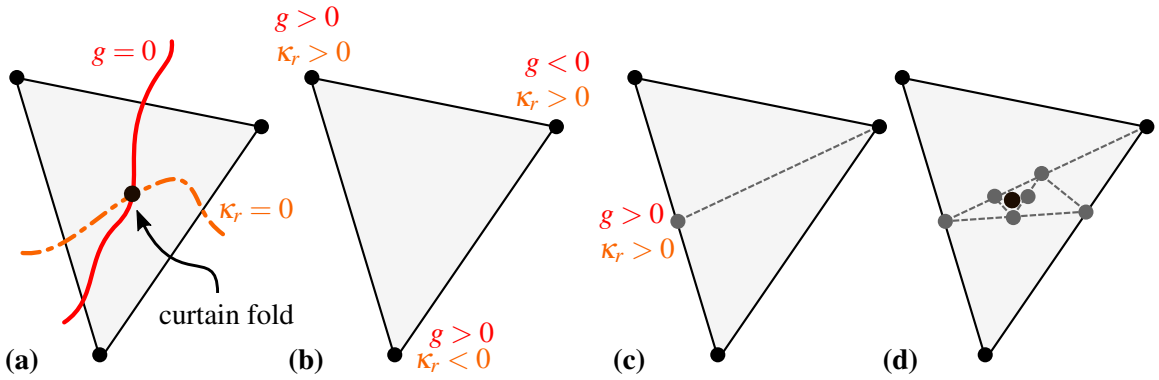


Figure 7.10: Contour curtain fold detection on smooth surfaces — **(a)** Given a base mesh triangle, we wish to find all points that satisfy both $g(\mathbf{u}) = 0$ and $\kappa_r(\mathbf{u}) = 0$. **(b)** We find triangles that contain sign changes of both functions between the vertices. **(c)** For each such triangle, we split the triangle into two, by bisecting the long edge, and recursing into each of these triangles. **(d)** When this process leads to a very small triangle with sign changes, a contour curtain fold is marked at the centroid of this triangle.

For parametric surfaces, radial curvature can be computed directly from the definition.

Contour curtain fold detection. Detecting curtain folds on smooth surface contours entails finding surface points where both $g(\mathbf{u}) = 0$ and $\kappa_r(\mathbf{u}) = 0$. The simplest approach is to perform linear interpolation within a face, as in Section 6.2.3. However, this may not be sufficiently accurate.

A more precise procedure is as follows (Bénard et al., 2014). The algorithm first searches for triangles where both the functions $g(\mathbf{u})$ and $\kappa_r(\mathbf{u})$ exhibit sign changes among the triangle vertices. When such a triangle is found, it is bisected along the edge that is longest in parameter ($\mathbf{u}$) space (Figure 7.10). This sign-change test and splitting process is repeated in the two new triangles. When the recursion detects a very small triangle with sign crossings in both functions, the centroid of that triangle is a curtain fold preimage location. (Note that the triangle splitting is not applied to the surface; the new triangles are stored only during this recursion and new vertex locations are computed by exact evaluation of $\mathbf{p}(\mathbf{u})$).

Curtain folds on other curves. Other types of curves may also have curtain folds. In general, for all curves, a curtain fold occurs when the 3D tangent to the curve is aligned to the view vector. This implies that the surface is normal to the view vector. Hence, at a curtain fold, the curve also intersects a contour generator. Hence, curtain folds do not need to be specially handled for non-contour curves, because they will be detected as curve-contour intersections.

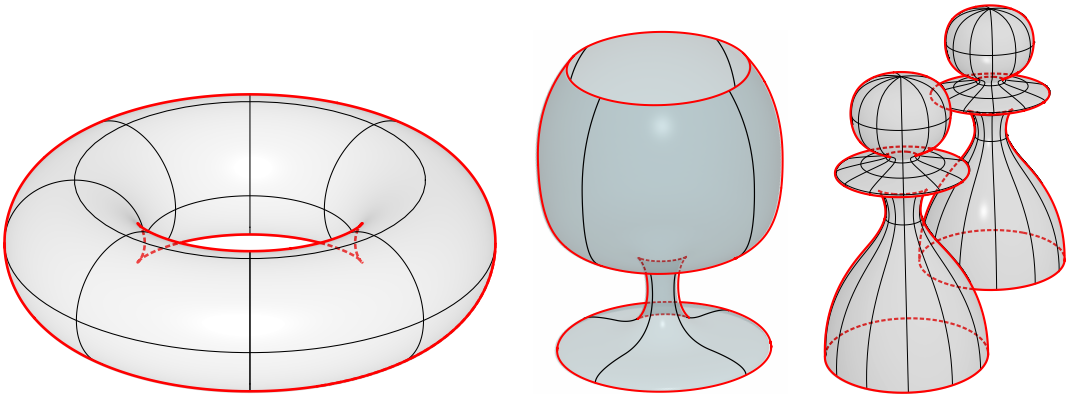


Figure 7.11: Results of Elber and Cohen (1990)'s patch-based contour extraction algorithm — Images generated with the IRIT modeling environment (Elber, 2018).

7.6 Visibility computation

Determining the visibility of the smooth surface contour is a significant, and very challenging problem. For mesh contours, the algorithms can rely on exact computations (Section 4.6), up to numerical precision. For example, a simple ray test can be used to determine the visibility of any point. On the other hand, performing an exact ray test for a parametric surface's contour involves a computation that is numerically unstable, since the true contour lies exactly on the boundary between visible and invisible. Hence, the mesh processing algorithms cannot be directly applied.

Previous authors have applied four different strategies to visibility for smooth surface contours. The first, described in Chapter 6, is to convert the surface to a mesh, and use heuristics (such as Interpolated Contours) to clean up the contours; this approach is simple, but can exhibit artifacts. The other three strategies are described next.

We also note that, in the past few decades, new methods for ray-tracing subdivision surfaces have been developed, e.g., (Kobbelt et al., 1998; Tejima et al., 2015; Benthin et al., 2015), and it may be time to revisit their usefulness for this problem.

7.6.1 Ray-casting the smooth surface

The first approach is to directly apply ray-casting and singularity detection on the smooth surface, generalizing the procedure for meshes. Because ray tests are unstable on the contour, one may perform them elsewhere on the surface, and then propagate visibility based on image-space relationships between curves. Elber and Cohen (1990) perform those tests along a subset of isoparametric curves. While this method is demonstrated for simple surfaces (Figure 7.11), there are a few theoretical issues that suggest it may not work robustly for general surfaces. First, it assumes that all visible contours touch one of the isoparametric curves, which may not always be the case for complex models. Second, propagating visibility information depends on robustly computing image-space intersections

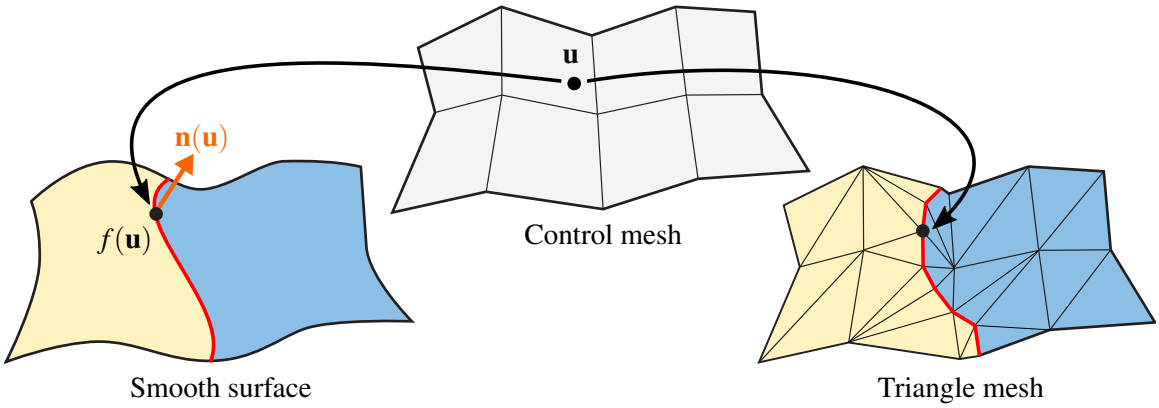


Figure 7.12: Contour-Consistent tessellation — Each vertex on the output triangle mesh produced by Bénard et al. (2014) corresponds to a preimage point $\mathbf{u}$ on the control mesh that maps to a point $f(\mathbf{u})$ on the smooth surface. The orientation (front- or back-facing) of each face of the triangle mesh is consistent with the smooth surface orientation $g(\mathbf{u})$.

between smooth curves. Once again, numerical instabilities are likely to arise, since curves may often be nearly tangent to each other in projection. As will all visibility propagation schemes, a single visibility computation error can propagate to create many erroneous visibility errors, such as a large silhouette curve that disappears.

One can imagine more robust versions of this procedure. To our knowledge, this strategy has not been explored since Elber and Cohen (1990).

7.6.2 Planar Maps

Winkenbach and Salesin (1996) used a Planar Map for visibility computations and stylization. A Planar Map can potentially ensure a consistent topology, even in the presence of errors, and allows more control over stylization of the regions.

Their Planar Map was computed from a mesh tessellation of the surface. However, they also numerically refine the contours separately from the Planar Map, and their contours might not exactly match the Planar Map visibility, potentially causing small visibility errors. Following the work of Gangnet et al. (1989), they restrict all the edge endpoints to have integer coordinates and use infinite-precision rational arithmetic to compute exact intersections which may mitigate some of the mismatch.

To our knowledge, Planar Maps for true smooth surface representations have not further been explored since Winkenbach and Salesin (1996).

7.6.3 Contour-consistent tessellation

Finally, we summarize an approach that we developed, in collaboration with Michael Kass (Bénard et al., 2014). Our approach is, for a given viewpoint, to tessellate the smooth surface into a triangle mesh for which the contour generators have the same topology as they do for

(a) Contours **(b)** Chained curves **(c)** Closeup view on **(b)**

Figure 7.13: Contour-consistent contours of a smooth surface (Bénard et al., 2014) — Compare with the contours in Figures 6.1 and 6.4. **(a)** Contours computed with the contour-consistency algorithm correctly represent the contours of the original smooth surface. **(b)** Chaining these segments gives smooth, coherent curves. **(c)** Visibility is well-defined for these curves. "Red" © Disney/Pixar

the original smooth surface. The triangle mesh is also geometrically close to the smooth surface. Because we have effective and exact algorithms for contour rendering on meshes (Chapters 3 and 4), this guarantees a valid contour rendering, approximating that of the smooth surface.

The algorithm creates a new mesh initialized by copying the smooth surface's control mesh (Figure 7.12). Throughout, the algorithm maintains a pointwise correspondence between the smooth surface and its polygonal approximation. We provide conditions that can be used to prove that the new mesh has achieved topologically-equivalent contours. The main goal is that front-faces on the mesh should correspond to front-facing regions on the surface, and back-faces should correspond to back-facing regions. The algorithm performs a sequence of local transformations on the mesh until these conditions are met.

The method still does not have all the formal guarantees that one would like; in principle, there are a few ways we discuss where it could go wrong. These do not seem to be a problem in practice. However, it is complex to implement and the computation is slow. Improving this is an area for future work.

This method produces a triangle mesh whose contour edges are topologically equivalent and geometrically close to the contour generators of the smooth surface (Figures 7.6b and 7.13). Hence, it is the only current method that computes accurate contours and visibility for smooth surfaces, avoiding flickering artifacts when animated (Figure 7.14).

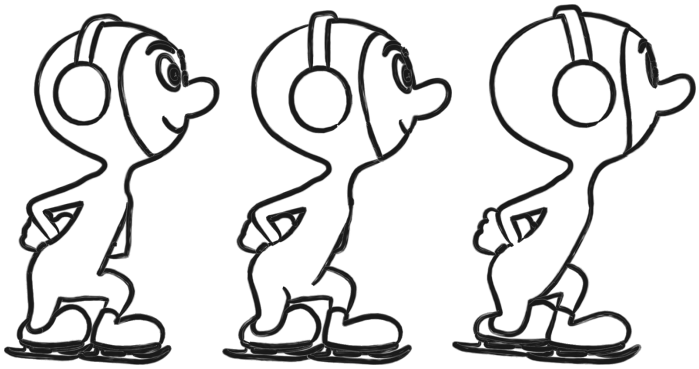


Figure 7.14: Contour-consistent contours stylized with tapered strokes (Bénard et al., 2014) — Compare with the contours in Figure 6.5. Contour-consistent contours do not suffer from breaks and gaps, producing more coherent animated strokes. "Red" ©Disney/Pixar

8

IMPLICIT SURFACES: CONTOURS AND VISIBILITY

We now survey contour extraction and visibility algorithms for implicit surfaces. Implicit surfaces are smooth surfaces, so much of the theory from the previous section applies to them, but implementation is different. Implicit surfaces are used less frequently in computer graphics, but are often used in 3D imaging and for other kinds of volumetric data.

8.1 Surface definition

An implicit surface $\mathscr{S}$ is defined as the isocontour of a scalar function $f : \mathbb{R}^3 \to \mathbb{R}$,

$$\mathscr{S} = \{\mathbf{p} \in \mathbb{R}^3 | f(\mathbf{p}) = \rho\},$$

where $\rho \in \mathbb{R}$ is a target *isovalue*. It is thus also called an *isosurface*. An implicit surface is well-defined if f does not have any *singular* points, i.e., its gradient ∇f is defined and non-zero everywhere. The isosurface forms a 2D manifold, partitioning the space into itself and two connected open sets: the *interior*, where $(f - \rho) < 0$, and the *exterior*, where $(f - \rho) > 0$ by convention.

The surface normal at $\mathbf{p}$ is the normalized gradient of f:

$$\mathbf{n} = \frac{\nabla f(\mathbf{p})}{\|\nabla f(\mathbf{p})\|},$$

where

$$\nabla f(\mathbf{p}) = \left(\left.\frac{\partial f}{\partial x}\right|_{\mathbf{p}}, \left.\frac{\partial f}{\partial y}\right|_{\mathbf{p}}, \left.\frac{\partial f}{\partial z}\right|_{\mathbf{p}} \right)^{\top}.$$

The normal curvature in any tangent space direction $\mathbf{t}$ is given by:

$$\kappa_{\mathbf{n}}(\mathbf{t}) = \frac{\mathbf{t}^{\top}(\nabla \mathbf{n})\mathbf{t}}{\|\mathbf{t}\|^2}$$

with $\nabla \mathbf{n}$ the gradient of the normal, i.e., the projection of the normalized Hessian (matrix of second partial derivatives) $\mathrm{H}f = \nabla^2 f$ onto the tangent plane.

Many approaches have been taken to compute the contour generator.

8.2 Contour extraction

The contour generator $\mathscr{C}$ of an implicit surface $\mathscr{S}$ seen from a camera center $\mathbf{c}$ is defined as the set of points $\mathbf{p}$ such that:

$$\begin{aligned} & (f(\mathbf{p})-\mathbf{c})\cdot\mathbf{n}=0 \\ \Leftrightarrow \quad & (f(\mathbf{p})-\mathbf{c})\cdot\nabla f(\mathbf{p})=0, \end{aligned}$$

which is itself an implicit function.

8.2.1 Contour tracing

The most basic contour tracing algorithm is the generic algorithm described by Dobkin et al. (1990) for tracing the contour of any smooth function from $\mathbb{R}^n$ to $\mathbb{R}^k$ ($k<n$). A drawback of this method is that it only extracts a fixed resolution piecewise-linear approximation of the contour. If the implicit function f is at twice continuous, we can directly work with the function f and trace an approximation of the contour based on its analytical tangent vector by numerical integration (Bremer and Hughes, 1998; Foster et al., 2005; Plantinga and Vegter, 2006).

Assuming (for simplicity) an orthographic projection along the direction $\mathbf{v}$, a parametric curve $c:\mathbb{R}\to\mathbb{R}^3$ lies on the contour generator of an implicit surface $\mathscr{S}$ if:

$$\begin{cases} f(c(t))=0 \\ \mathbf{v}^\top\nabla f(c(t))=0.^{1} \end{cases} \tag{8.1}$$

Denoting $\mathbf{w}=c'(t)$ the tangent vector of the curve and differentiating each equation with respect to t, we get:

$$\begin{cases} \nabla f(c(t))\cdot\mathbf{w}=0 \\ \mathbf{v}^\top \mathrm{H}f(c(t))\mathbf{w}=0. \end{cases}$$

This implies that $\mathbf{w}$ is proportional to the cross-product of the gradient at its basepoint and the product of the Hessian at the basepoint with the view direction, that is:

$$\mathbf{w}\propto\nabla f(c(t))\times\mathbf{v}^\top\mathrm{H}f.$$

If we know a starting point $\mathbf{p}_0$ on the contour generator, we can progressively trace the full curve by taking small steps in the direction of the tangent $\mathbf{w}$. This corresponds to a numerical Euler integration scheme where, at each step:

$$\begin{aligned} \mathbf{p}_{i+1} &= \mathbf{p}_i+\varepsilon F(\mathbf{p}_i), \\ \text{with}\quad F(\mathbf{p}) &= \nabla f(\mathbf{p})\times\mathbf{v}^\top\mathrm{H}f(\mathbf{p}). \end{aligned}$$

[1] Recalling that $\mathbf{a}\cdot\mathbf{b}=\mathbf{a}^\top\mathbf{b}$.

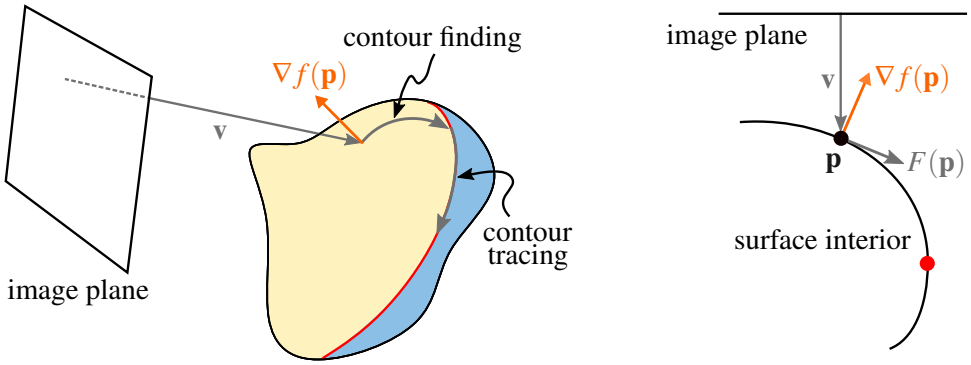


Figure 8.1: Contour finding — To find a starting position for tracing the occluding contour, Bremer and Hughes (1998) shoot random rays in the view direction $\mathbf{v}$ from the image plane (left), and march along the surface in the tangent direction aligned with the projection of the gradient (right) until $\mathbf{v} \cdot \nabla f(\mathbf{p})$ changes sign.

for small step size ε. Because the tangent vector at a curtain fold vanishes, tracing stagnates when it reaches a curtain fold.

The full process can be summarized as follows:

- Find a starting point on the contour generator.
- Trace out the contour curve by Euler integration.
- Stop when the curve returns to the starting point or stagnates.
- If it stagnates, return to the starting point and trace in the opposite direction.

Stabilized integration. Euler integration is known to be unstable, that is, the position $\mathbf{p}$ might quickly leave the contour, even with a small step sizes, in critical configurations. To improve convergence, two correction terms can be added to the vector field $F(\mathbf{p})$. The first correction enforces the computed position to lie on the implicit surface by pointing towards the surface at all point of space:

$$F_{\text{surface}}(\mathbf{p}) = \frac{-f(\mathbf{p})\nabla f(\mathbf{p}))}{\|\nabla f(\mathbf{p})^2\|}.$$

The second correction ensures that the integration follows the contour generator. In the same way as $-f\nabla f$ tends to drive f to zero, $-g\nabla g$ with $g(\mathbf{p}) = \mathbf{v} \cdot \nabla f(\mathbf{p})$ tends to drive g to zero, i.e., towards the contour generator, leading to:

$$F_{\text{contour}}(\mathbf{p}) = \frac{-(\mathbf{v} \cdot \nabla f(\mathbf{p}))\mathbf{v}^\top \mathrm{H}f(\mathbf{p})}{\|\nabla f(\mathbf{p})^2\|}.$$

The final Euler step is simply the weighted sum of the vector fields:

$$\begin{aligned}\mathbf{p}_{i+1} &= \mathbf{p}_i + \varepsilon\left(F(\mathbf{p}_i) + F_{\text{surface}}(\mathbf{p}_i) + kF_{\text{contour}}(\mathbf{p}_i)\right)\\ &= \mathbf{p}_i + \frac{\varepsilon}{\|\nabla f(\mathbf{p}_i)\|^2}(\nabla f(\mathbf{p}_i) \times \mathbf{v}^\top \mathrm{H}f(\mathbf{p}_i) - f(\mathbf{p}_i)\nabla f(\mathbf{p}_i)))\\ &\quad - k(\mathbf{v}\cdot\nabla f(\mathbf{p}_i))\mathbf{v}^\top \mathrm{H}f(\mathbf{p}_i))\end{aligned}$$

with k a user-defined scalar value — Bremer and Hughes (1998) recommend choosing $k = 0.5$. With those correction terms, the vector field does not vanish at curtain folds anymore, which may prove problematic if the tracer overshoots.

Finding starting points. Different approaches have been proposed to find starting positions on the contour generator. Bremer and Hughes (1998) shoot random rays from the orthographic camera and, when an intersection is found, they march along the surface in a tangent direction whose image-space projection is in the same direction as that of the gradient, i.e.,

$$F(\mathbf{p}) = \frac{\nabla f(\mathbf{p}) \times (\mathbf{v} \times \nabla f(\mathbf{p}))}{\|\nabla f(\mathbf{p})\|^2},$$

by Euler integration, until the sign of $\mathbf{v}\cdot\nabla f(\mathbf{p})$ changes (Figure 8.1). The term F_{surface} can be added to the vector field to ensure that the integrated positions remain close to the surface.

Instead of casting rays each time the viewpoint changes, Foster et al. (2005) precompute a dense set of seed points using the surface-constrained "floater" particles of Witkin and Heckbert (1994), and select the points that are close enough to the contour generator, i.e., such as those where $|\mathbf{v}\cdot\nabla f(\mathbf{p})|$ is below a threshold.

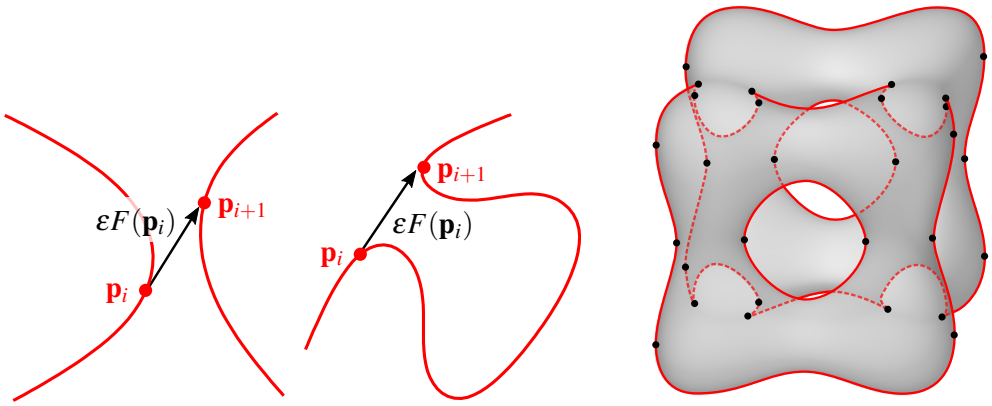
8.2.2 Extraction as surface-surface intersection

Equation 8.1 can be interpreted slightly differently: the contour generator can be viewed as the curve at the intersection of two implicit surfaces, the object surface $\mathscr{S}$ and the *contour surface* (Figure 8.3) defined implicitly as the zero-set of $\nabla f(\mathbf{p})\cdot\mathbf{v}$. (Stroila et al. (2008) called it the "silhouette surface.")

To delineate this intersection, Stroila et al. (2008) simultaneously constrain the "floater" particles of Witkin and Heckbert (1994) to lie on the implicit surface $\mathscr{S}$ and on the contour surface. After optimization, the particles can be connected together to form closed loop on the implicit surface. The differential properties of the contour curve can be leveraged to properly select each particle neighbors, although this does not guarantee accurately tracing the contour, nor finding all contours.

8.2.3 Extraction on a mesh

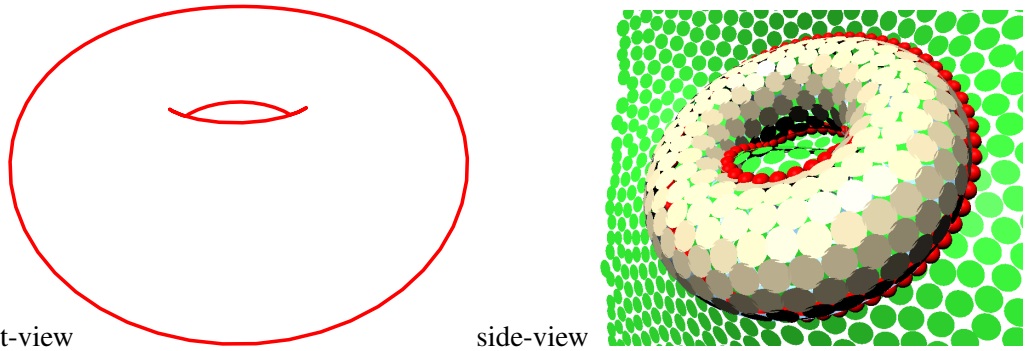
A even simpler, but very approximate, approach consists in first extracting a polygonal mesh from the implicit surface, e.g., with the Marching Cubes algorithm (Lorensen and



(a) fixed step size issues

(b) contours traced with a dynamic step size

Figure 8.2: Implicit surface contour generator tracing — With a fixed step size **(a)**, the traced contour may jump to another component of the contour (left) or skip a part of it (right). **(b)** With a dynamic step size integration scheme, Plantinga and Vegter (2006) accurately trace the contour generator of a complex implicit tangle cube: $x^4 - 5x^2 + y^4 - 5y^2 + z^4 - 5z^2 + 10 = 0$. The starting points are indicated by the black dots.



front-view side-view

Figure 8.3: Surface-surface intersection — The apparent contour (left) along the view direction $\mathbf{v}$ is the projection of the contour generator (in red), shown from a side-view (right), at the intersection of the implicit surface $\mathscr{S}$ (in yellow) and the implicit contour surface $\nabla f \cdot \mathbf{v} = 0$ (in green). Images generated with the "Wickbert" particles library (Stroila et al., 2011).

Cline, 1987) and its extensions (de Araújo et al., 2015), computing its interpolated contours (Section 6.2), and projecting those onto the implicit surface after view-dependent subdivision (Schmidt et al., 2007) (Figure 8.4). Since the implicit function f is defined everywhere in $\mathbb{R}^3$, the projection of a point $\mathbf{p}$ on the implicit surface $f(\mathbf{p}) = \rho$ can be computed by walking along the gradient of f, updating $\mathbf{p}$ with the following convergence iteration:

$$\mathbf{p} \leftarrow \mathbf{p} + \frac{(f(\mathbf{p}) - \rho)\nabla f(\mathbf{p})}{\|\nabla f(\mathbf{p})\|}.$$

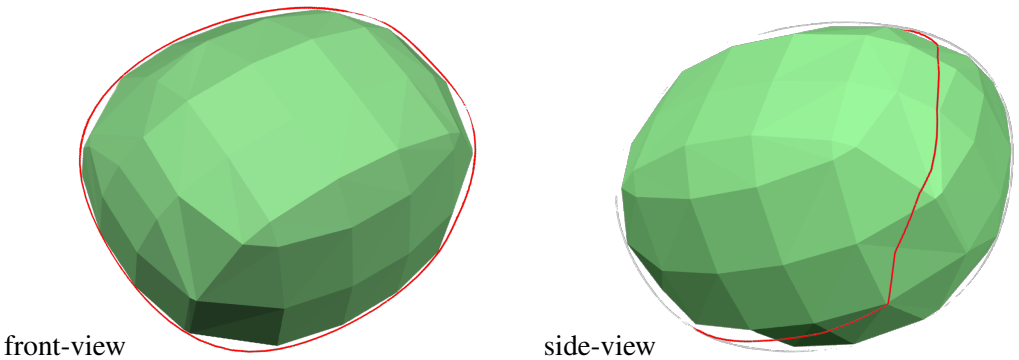


Figure 8.4: Proxy-based method — The smooth surface occluding contour is approximated by computing the interpolated contours of on a coarse base mesh, projecting the vertices of this coarse contour onto the actual smooth surface and subdividing it. Images generated with "ShapeShop"(Schmidt, 2008).

This scheme is a form of gradient descent on $(f(\mathbf{p})-\rho)^2$. It usually leads to the isovalue after a few iterations if f is smooth enough, though it may get stuck when there are discontinuities.

8.2.4 Extraction on a mesh

A even simpler, but very approximate, approach consists in first extracting a polygonal mesh from the implicit surface, e.g., with the Marching Cubes algorithm (Lorensen and Cline, 1987) and its extensions (de Araújo et al., 2015), computing its interpolated contours (Section 6.2), and projecting those onto the implicit surface after view-dependent subdivision (Schmidt et al., 2007) (Figure 8.4). Since the implicit function f is defined everywhere in $\mathbb{R}^3$, the projection of a point $\mathbf{p}$ on the implicit surface $f(\mathbf{p})=\rho$ can be computed by walking along the gradient of f, updating $\mathbf{p}$ with the following convergence iteration:

$$\mathbf{p} \leftarrow \mathbf{p} + \frac{(f(\mathbf{p})-\rho)\nabla f(\mathbf{p})}{\|\nabla f(\mathbf{p})\|}.$$

This scheme is a form of gradient descent on $(f(\mathbf{p})-\rho)^2$. It usually leads to the isovalue after a few iterations if f is smooth enough, though it may get stuck when there are discontinuities.

8.3 Visibility

As with mesh contours, two families of approaches can be used to determine the visibility of the extracted contour generators. The first one is based on ray-casting and thus more accurate but only suitable for offline computation. The second one uses the depth buffer algorithm and is well suited for real-time applications, even though the implicit nature of the surface makes the creation of the depth buffer more complex.

Ray-casting and propagation. The simplest, but most time-consuming, method consists in testing the visibility of every point $\mathbf{p}_i$ on the discretized occluding contour by casting a ray from the camera towards $\mathbf{p}_i$ and performing a ray-surface intersection test. Due to numerical issues, $\mathbf{p}_i$ may not exactly lie on the implicit surface; therefore it may be locally occluded by the surface leading to a spurious ray intersection. To mitigate this problem, Bremer and Hughes (1998) discard surface intersections that are too close to $\mathbf{p}_i$ which, in turn, may reveal contours that are barely obscured by nearby pieces of surface.

To reduce the number of ray tests, one can first check every n^{th} point for occlusion, and then refine by testing the intermediate points when the visibility changes between $\mathbf{p}_i$ and $\mathbf{p}_{i+n-1}$ (Bremer and Hughes, 1998; Foster et al., 2005). To reduce the number of tests even further, a view graph can be built and the visibility information can be propagated along the contour chains (Section 4.1). However the propagation rules slightly differ from those for polygonal meshes since implicit surfaces are closed by construction. To reduce the number of ray-tests further, Stroila et al. (2008) describe methods to propagate Quantitative Invisibility (Section 4.6) on contours of implicit surfaces.

Depth buffer. The alternative solution is to render the 3D scene into a depth buffer, and then to use this buffer to determine the visibility of the apparent contour. Besides the problems mentioned in Section 5.3.1, an additional difficulty of such an approach is that, unlike polygonal meshes, implicit surfaces cannot be rasterized directly. One could again extract a polygonal mesh from the implicit surface, but a highly refined tessellation is required to avoid visual artifacts – otherwise the mesh contours will largely disagree with the smooth contours – which is computationally expensive.

Instead, we can discretize the implicit surface using *surfels* (Pfister et al., 2000), oriented ellipses that are traditionally used to render 3D point clouds. For instance, Foster et al. (2005) generate a surfel for every "floater" particle distributed on the implicit surface and orient them according to the surface gradient. Each surfel is then projected into the 2D image plane and rasterized with depth writes activated. To provide accurate results, the implicit surface needs to be suitably covered by surfels, which may require a large number of particles in areas of high curvature (Meyer et al., 2005).

On deforming surfaces, dynamically recomputing such particle distribution is too costly for real-time applications. In the same spirit as the painterly rendering technique of Meier (1996), Schmidt et al. (2007) first distribute surfels on a low resolution mesh (previously used for contour extraction), and then project them onto the implicit surface. However, after projection, the surfels may not properly cover the surface. Even thought a simple heuristic is proposed to non-uniformly scale them, it cannot guarantee that the surface will be accurately approximated by the distribution of surfels.

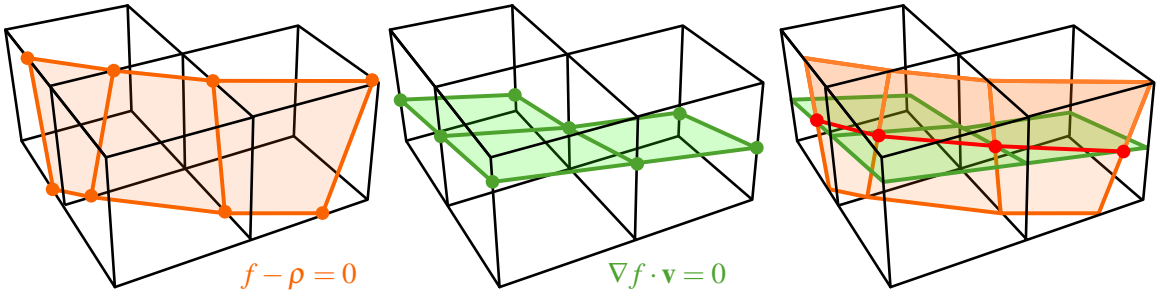


Figure 8.5: Marching line algorithm — The intersections of the isosurface (left) and contour surface (center) with each face of every voxel are first computed. The resulting segments are intersecting on the faces at contour points (right).

8.4 Volumetric data

Volumetric data can be seen as a special case of implicit surfaces, one where the implicit function f is discretized on a regular 3D grid (voxel grid) into density values:

$$v_{ijk} = f(\mathbf{p}_{ijk}) = f(x_i, y_j, z_k). \tag{8.2}$$

Different methods make different assumptions on the density between these values. In this case, the local curvature and scale of small features is limited by the discretization, making it possible to make better guarantees about contour and intersection detections, unlike with arbitrary implicit surfaces.

There are two general approaches to contour visualization. The first entails defining an isosurface, and then extracting contours of the isosurface, similar to what was done in the general implicit surface case. The second approach is analogous to image-space contour rendering (Chapter 2): it uses a conventional volume rendering method, but with transfer functions designed to emphasize contours

8.4.1 Isosurface extraction

To compute the contours corresponding to a given iso-value ρ, we could first extract the corresponding iso-surface $f(\mathbf{p}_{ijk}) = \rho$ with, e.g., the Marching Cubes algorithm (Lorensen and Cline, 1987), compute its normals as $\mathbf{n} = \nabla f$, and extract the surface Interpolated Contours (Section 6.2). However, as discussed previously for implicit surfaces, occluding contours can be seen as the zero-set of two implicit functions:

$$\begin{cases} f(\mathbf{p}_{ijk}) - \rho = 0 \\ \nabla f(\mathbf{p}_{ijk}) \cdot \mathbf{v} = 0 \end{cases}$$

or, geometrically, as the intersection of two implicit surfaces.

The simplest solution to extract a piecewise linear approximation of the intersection curves of these implicit functions is the marching lines algorithm (Thirion and Gourdon,

1996; Burns et al., 2005). For each voxel, this method first computes the line segments at the intersection of both functions with the voxel faces, using linear interpolation of the density and gradient values at the voxel corners (Figure 8.5, left and center). It then finds the intersection points between these two set of lines on each face. These contour points are eventually connected to produce contour chains (Figure 8.5, right).

The piecewise linear approximation of both the implicit functions and their intersection might be too crude. Assuming that the input scalar field is smooth enough, Schein and Elber (2004) use trivariate B-spline functions to model the volumetric data. The scalar values are used as control points of a trivariate tensor product B-spline function $D : \mathbb{R}^3 \to \mathbb{R}$:

$$D(u,v,w) = \sum_i \sum_j \sum_k f(\mathbf{p}_{ijk}) N_i^d(u) N_j^d(v) N_k^d(w),$$

where $N_i^d(u)$, $N_j^d(v)$ and $N_k^d(u)$ are the B-spline basis functions of degree d that controls the smoothness of the representation. Extracting the contour then boils down to resolve the following system of equations:

$$\begin{cases} D - \rho = 0 \\ \nabla D \cdot \mathbf{v} = 0 \end{cases}$$

The multidimensional Newton-Raphson solver of Elber and Kim (2001) can be used to solve this system to a desired accuracy starting from a dense set of seed points. If quadratic or higher basis functions are used, the gradient field of the trivariate B-spline function ∇D is continuous, and thus the contours are smooth and continuous. In addition, since the solver can refine the data at any parametric location (u,v,w) and not just discrete grid points, the contour approximation is better adapted to the input data. However, with this method, the connectivity of the extracted contour points cannot be trivially inferred for the voxel grid anymore, limiting the rendering and stylization possibilities.

Acceleration strategies. A naive implementation of the two previous approaches would have an $O(n^3)$ complexity for a dataset of size $n \times n \times n$ voxels. In practice, a typical isosurface will have surface area $O(n^2)$ and thus its contour will have size $O(n)$ edges (Section 3.6).

To speed-up contour extraction in volumes, strategies similar to those used for mesh contours can be employed.

Burns et al. (2005) adapt randomized search with temporal coherence (Section 3.7.3). They first test random voxels until a contour is found. Then, they move to the voxel adjacent to the face containing one of the intersection points and repeat until the initial voxel is reached, forming a contour loop. The random sampling strategy to find a starting voxel can be further improved by leveraging temporal coherence, i.e., searching nearby contour-containing voxels from the previous frame, and by gradient descent, alternatively walking along the gradient of the isosurface and contour functions until a new contour-containing voxel is found.

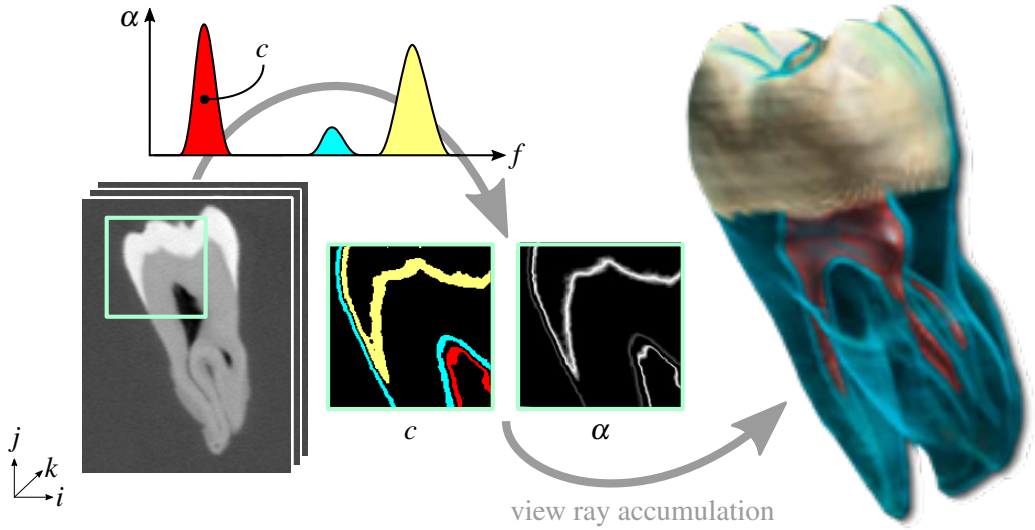


Figure 8.6: Direct volume rendering (Ikits et al., 2004) — The density value f of each voxel p_{ijk} is mapped through a transfer function (top) into a color c and an opacity α; those are then accumulated along each view ray to produce an image of the data.

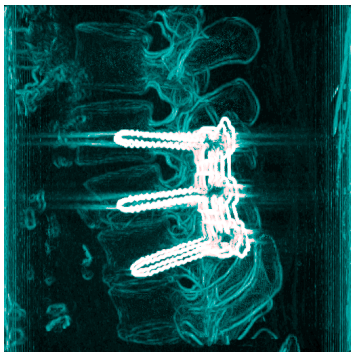
An acceleration data-structure can be built during a preprocessing step. Elber and Kim (2001) construct a 2D lookup table whose first dimension corresponds to isovalue ranges and second dimension corresponds to bounding cones covering the unit sphere. The trivariate B-spline function D is then clustered into "singletons" based on its isovalue and gradient, and stored them in the table. At runtime, given a view direction $\mathbf{v}$ and isovalue ρ, only the relevant singletons can be efficiently retrieved from the table, defining the seed points for the Newton-Raphson solver.

Visibility. Finally, the visibility of the contour can be computed with respect to the target isosurface by tracing a ray from each contour point towards the camera, similarly to Bremer and Hughes (1998). For each voxel traversed by the ray, we need to test whether the isosurface is intersected. It is achieved by checking if the sign of f changes at the intersection of the ray with the face by which it leaves the voxel. If it does, the ray passes from outside to inside the isosurface, and the contour point is thus occluded.

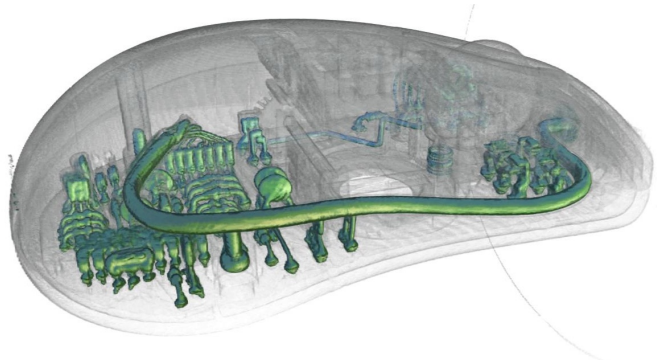
8.4.2 Direct volume rendering

In this approach, we use conventional volume rendering, and modify the transfer function to visualize contours.

In conventional volume visualization (Kaufman and Mueller, 2005), the initial density value $v_{ijk} = f(\mathbf{p}_{ijk})$ of each voxel $\mathbf{p}_{ijk}$ of the discrete volume is first mapped through a transfer function into a color value c_{ijk} and an opacity α_{ijk}. Rays are then cast from each pixel of the camera, along which colors and opacities are regularly sampled with appropriate interpolation (often tri-linear) in the voxel grid. Those samples are eventually composited in



(a) Vertebra rendered with the transfer function of Csébfalvi et al. (2001).



(b) Computer mouse visualized with the interactive technique of Lum and Ma (2002).

Figure 8.7: Contour enhanced direct volume rendering.

front-to-back order to yield a single color per pixel (Figure 8.6). For shading computation, a normal at each voxel is computed as the normalized gradient of f, which is usually approximated with central differences:

$$\nabla f(\mathbf{p}_{ijk}) = \nabla f(x_i, y_j, z_k) \approx \frac{1}{2} \begin{bmatrix} f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k) \\ f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k) \\ f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1}) \end{bmatrix},$$

although more advanced gradient estimation operators have been proposed (Lichtenbelt et al., 1998). This approach is called the "pre-classified model" since voxel densities are mapped to colors and opacities prior to interpolation. An alternative solution is the "post-classified model" that interpolates the voxel densities before mapping the resulting values to colors and opacities, and thus tends to produce sharper results.

To enhance occluding contours, we can increase the opacity of voxels whose gradient is near perpendicular to the view direction $\mathbf{v}$. Given input opacity α_{ijk}, Ebert and Rheingans (2000) suggest using the following opacity transfer function:

$$\alpha'_{ijk} = \alpha_{ijk}(k_{sc} + k_{ss}(1 - |\nabla f(\mathbf{p}_{ijk}) \cdot \mathbf{v}|^{k_{se}}),$$

where k_{sc} controls the scaling of non-contour regions, k_{ss} controls the amount of contour enhancement, and k_{se} controls the sharpness of the contour "curve".

When one wishes to only visualize data variations and ignore scalar values, voxel color can directly be defined based on the gradient magnitude. For instance, Csébfalvi et al. (2001) propose the following color transfer function:

$$c_{ijk} = w(|\nabla f(\mathbf{p}_{ijk})|)(1 - |\nabla f(\mathbf{p}_{ijk}) \cdot \mathbf{v}|)^{k_{se}},$$

with w a windowing function selecting the range of interest in the gradients. Opacity modulations can also be computed with the same transfer function and combined with colors with standard front-to-back compositing (Figure 8.7a).

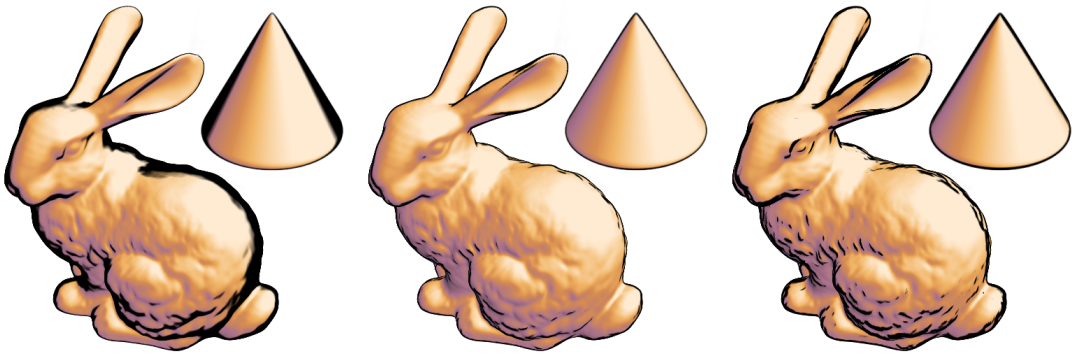


Figure 8.8: Contour thickness variations in volume rendering — From left to right, apparent contours based solely on $\mathbf{v} \cdot \mathbf{n}$, with controlled thickness using $T = 1$ and $T = 2.5$. Images rendered with "miter" courtesy of Gordon Kindlmann, more information at http://www.sci.utah.edu/~gk/vis03/.

Alternatively, maximum intensity projection can be used. It consists in only keeping the sample with the highest intensity (color times opacity) along each ray. This tends to reduce the visual overload since a single (but potentially different) iso-value is represented per pixel, but discards any depth information, which may make the interpretation of the resulting image difficult. The depth ordering perception can be improved by local maximum intensity projection that selects the closest sample to the camera which is above a threshold.

Graphics hardware acceleration. The above methods involve expensive ray marching, which prohibits real-time rendering of large volumetric data sets. Graphics hardware can be used for acceleration (Lum and Ma, 2002; Nagy et al., 2002). In particular, the volume is sliced with polygons aligned with the view, and progressively accumulated in the image plane. Significant speed-ups are obtained by storing the voxel grid into a 3D texture and leveraging graphics hardware for the slicing and re-sampling operations. For contour rendering, the normalized gradient is stored as an additional 3D texture. It is accessed at each fragment of every slice and used to modulate the color and/or opacity of the fragment by its dot product with the view direction (Figure 8.7b).

Line thickness control. In the above methods, the thickness of the apparent contour varies in image-space (Figure 8.8 left). These thickness variations are related to the radial curvature κ_r of the selected iso-surfaces. For a given threshold on $|\mathbf{n} \cdot \mathbf{v}|$, the resulting contour is thicker on areas of low radial curvatures since the normal is nearly perpendicular to the view direction in this large, almost flat region. Conversely, in areas of high curvature, the normal quickly changes resulting on a very thin contour.

To solve this problem, Kindlmann et al. (2003) derive a 2D transfer function that takes the radial curvature κ_r into account and guarantees that the apparent contour will approximately have a constant, controllable thickness T in image-space (Figure 8.8 right).

To use this transfer function, we need to accurately estimate the curvature in the voxel grid and thus the Hessian of the 3D scalar field. Simply computing finite differences over the previously estimated normals would provide a very crude approximation. Instead, Kindlmann et al. (2003) use axis-aligned 1D continuous convolution filters for zero-, first- and second-derivative estimation. They show that the cubic B-spline and its derivatives provide a good tradeoff between accuracy and robustness to the noise.

8.4.3 Particle distribution

The particle-based implicit surface visualization technique of Foster et al. (2005) can also be adapted to volumes (Busking et al., 2008). During a pre-process, particles are distributed on a given isosurface by sampling the volume data with linear interpolation on a user-defined grid and applying some relaxation steps (Meyer et al., 2005). At runtime, particles whose normal is almost orthogonal to the view direction are selected for rendering. From each of those, a short line segment is traced in a direction locally parallel to the contour. A linear approximation of this direction can be computed as follows. In the local coordinate system formed by the two principal directions $\mathbf{e}_1$ and $\mathbf{e}_2$ and the the normal $\mathbf{n}$, the behavior of the normal can be linearly approximated by:

$$\tilde{\mathbf{n}}(u,v) = (-\kappa_1 u, -\kappa_2 v, 1)^\top,$$

with κ_1 and κ_2 the principal curvatures in the corresponding principal directions. The view vector expressed in the same coordinate frame is:

$$\tilde{\mathbf{v}} = (\mathbf{e_1}\cdot\mathbf{v}, \mathbf{e_2}\cdot\mathbf{v}, \mathbf{n}\cdot\mathbf{v})^\top.$$

The contour lime in this frame can thus be defined as the set of parametric locations (u,v) such as:

$$\tilde{\mathbf{v}}\cdot\tilde{\mathbf{n}}(u,v) = 0 \Leftrightarrow -\kappa_1(\mathbf{e_1}\cdot\mathbf{v})u - \kappa_2(\mathbf{e_2}\cdot\mathbf{v})v + \mathbf{n}\cdot\mathbf{v} = 0,$$

from which a parallel direction in world space can be derived:

$$\mathbf{d} = -\kappa_2(\mathbf{e_2}\cdot\mathbf{v})\mathbf{e}_1 + \kappa_1(\mathbf{e_1}\cdot\mathbf{v})\mathbf{e}_2.$$

In addition, to avoid thickness variations of the apparent contour (Figure 8.9), Busking et al. (2008) use a thresholding function that depends on the image-space distance T of the particle to the contour line approximation, i.e., assuming orthogonal projection:

$$T = \frac{(\mathbf{n}\cdot\mathbf{v})^2}{\sqrt{(\kappa_1(\mathbf{e_1}\cdot\mathbf{v}))^2 + (\kappa_2(\mathbf{e_2}\cdot\mathbf{v}))^2}}.$$

Since principal curvature information is view-independent, it can be pre-computed for all particles.

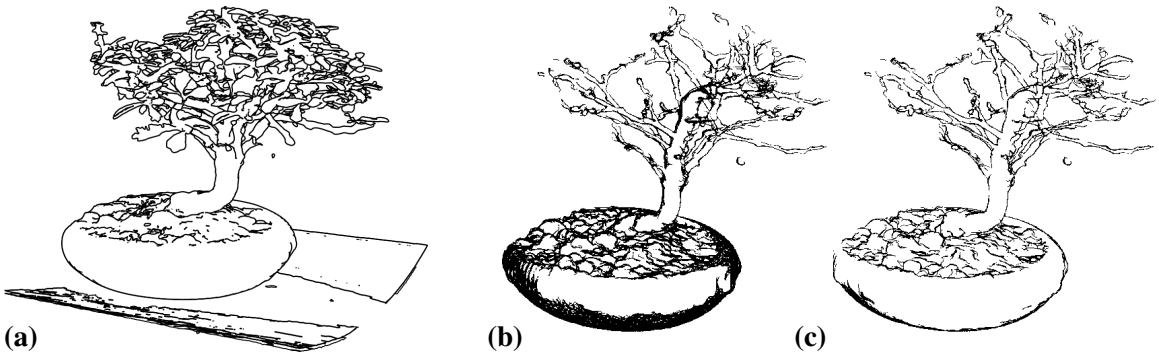


Figure 8.9: Bonsai dataset (256^3 voxels) — Contours extracted with the marching line method of Burns et al. (2005) **(a)** and the particle system of Busking et al. (2008) without **(b)** and with **(c)** thickness control. Volumetric dataset courtesy of Stefan Roettger (2012).

9 Stylized Rendering and Animation

We now come to the reward for all the hard work of curve extraction: rendering these curves in an attractive, artistic style. This chapter describes algorithms for stylized rendering and animation.

The basic steps, summarized in Figure 9.1, are to combine visible contour line segments into longer curves (Section 9.1), and then to render those curves with stroke textures, such as a pen, pencil, or paint strokes (Section 9.2). In many cases, it will be necessary to simplify the curves before rendering, for example, to remove unnecessarily details that an artist would never draw (Section 9.3).

Together with the strokes, we usually wish to draw the model with shading or texturing. We briefly survey shading and texturing methods in Section 9.4.

Finally, when rendering these models in animation, we often want to render strokes with coherent stylization over time. We survey coherent stylization in Section 9.5.

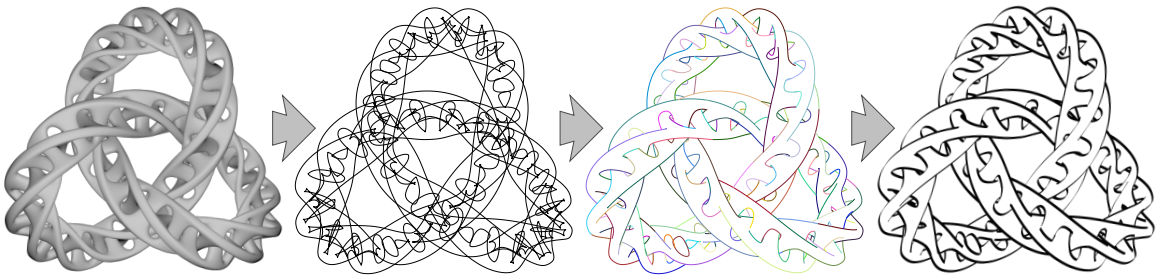


Figure 9.1: Stylized contour rendering pipeline — Starting from a 3D geometry, the contour line segments are first extracted. Their visibility is then computed by building their View Graph; here the visible chains are shown, each with a separate color. Topological simplification can be optionally applied for legibility or artistic purposes. Finally, each chain of the View Graph is rendered with stylized strokes. Images generated with Blender Freestyle.

9.1 Stroke extraction

The first stage of processing is to extract smooth curves from the View Graph (Section 4.5). As a reminder, the View Graph refers to the connected line segments extracted from the 3D model. Most line segments have pointers at each ends indicated which line segments they connect to. Some line segments are connected at singular points (Section 4.3), such as a T-junctions, where three visible segments and one invisible segment connect. Line segments between junctions must all have the same visibility. The line segment and singularity data structures record information about where they came from. For example, a line segment may record that it was extracted from a contour edges, and would include a pointer to the mesh edge that it came from. Some topological filtering on the View Graph may be necessary in order to extract clean strokes, such as by removing tiny loops. This filtering is described in Section 9.3.

One may also use a Planar Map instead of the View Graph (Winkenbach and Salesin, 1994, 1996; Eisemann et al., 2008). The Planar Map allows stylization to take into account the relationship between the curves and the shapes of the regions they enclose.

Chaining. From the View Graph, we need to extract a set of curves for stylization. These curves are simply chains of line segments, which can be smoothed and rendered. Normally, we will extract only the visible curves, but invisible curves can also be extracted for hidden-line renderings, in which the invisible curves are rendered in a different style from the visible curves. In the rest of this chapter, we will assume that only the visible curves are being extracted.

The basic approach to extracting these curves is greedy. We call this process "chaining." We pick some visible line segment at random, and follow pointers from one end of the line segment, following pointers from line segment to line segment, concatenating them into a list of line segments. If this chaining process returns to the original segment, then it is recorded as a closed loop. If the chaining process reaches a singularity, then the behavior depends on the singularity (refer to Figure 4.2). At a curtain fold cusp, chaining stops. At a T-junction, if the chain is in the foreground, then the chaining continues through the junction, otherwise it stops (Figure 9.2(a)). At a Y-junction, the chaining process continues through the chain to connect silhouette edges, and stops for boundary edges (Figure 9.2(b)).

The above process is repeated until all visible line segments have been added to a chain. Then, each chain can be rendered separately.

Depending on the rendering style, we may wish to use different chaining rules. For example, a style that renders a single thick silhouette around the entire object would extract a silhouette chain that follows the image-space object boundary. To do this, at each singularity, the chain follows whichever outgoing edge is on the silhouette. Determining whether a mesh edge is a silhouette edge could be determined with a ray test. Sousa and Prusinkiewicz (2003) explored other chaining strategies, and Grabli et al. (2010) allow the user to programmatically define how edges should be chained together. With this approach,

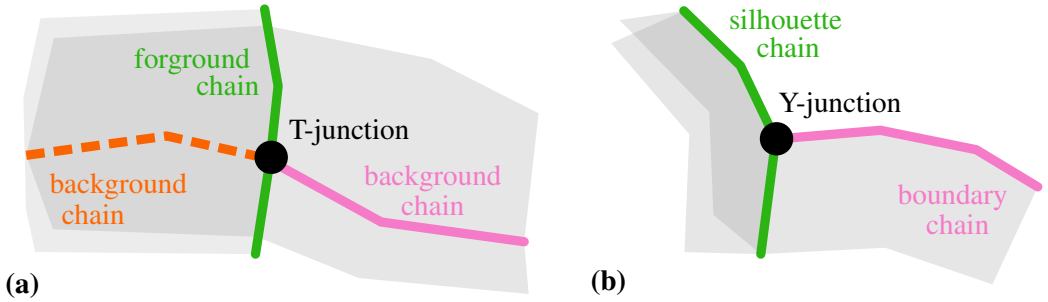


Figure 9.2: Segment chaining at junctions — **(a)** At a T-junction, a foreground chain continues through the junction, whereas a background chains stops. **(b)** At a Y-junction, a silhouette chain crosses the junction, whereas a boundary chain stops.

a given edge of the View Graph can even belong to multiple chains, allowing to produce sketchy drawing with overlapping strokes (Figure 9.4).

Smoothing. A chain is a polyline, i.e., a set of connected line segments. Converting a chain into a stroke typically entails smoothing polyline in some way. For example, one may merge redundant control points (e.g., control points that are fewer than 2 pixels apart), set a number of control points proportional to the stroke's arc-length, and then perform a least-squares B-spline fit. The spline can then be converted back to a list of finely-spaced control points. The specific amount of smoothing to apply is a stylistic choice.

The topology of the View Graph also must be taken into account when smoothing curves. For example, a common stylization would be to smooth every stroke independently. However, this can cause a T-junction to break, with the far stroke either penetrating the near stroke, or separating from it; note how the junction lines overlap in Figure 9.4(lower-right). This disconnection can be prevented either by constraining the smoothing algorithm or postprocessing it.

9.2 Stroke rendering

Once smooth strokes have been computed, they can be rendered. Strokes are typically parameterized with a skeletal stroke representation (Hsu and Lee, 1994). Skeletal strokes describe a parameterization of the region around the stroke. As illustrated in Figure 9.3, for each 2D vertex $\mathbf{v}_i$ of the stroke path, a "rib" vector $\mathbf{r}_i$ is constructed orthogonally to the direction defined by its previous $\mathbf{v}_{i-1}$ and next $\mathbf{v}_{i+1}$ vertices (if they exist):

$$\mathbf{r}_i = w_i \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \frac{\mathbf{v}_{i+1} - \mathbf{v}_{i-1}}{\|\mathbf{v}_{i+1} - \mathbf{v}_{i-1}\|},$$

scaled by the half thickness of the stroke w_i. Special treatments are required at places where the radius of curvature of the strokes is smaller than its half thickness, otherwise the ribs will cross each other producing folds, which is especially problematic at sharp corners since

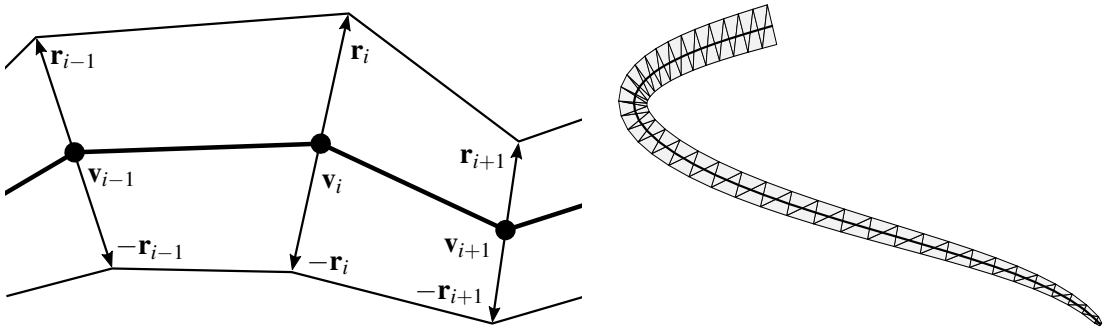


Figure 9.3: Skeletal stroke — For each vertex $\mathbf{v}_i$ of the stroke path, a rib vector $\mathbf{r}_i$ is generated along the angle bisector to give breadth to the stroke (left). The skeletal stroke can then be rendered as a triangle strip (right).

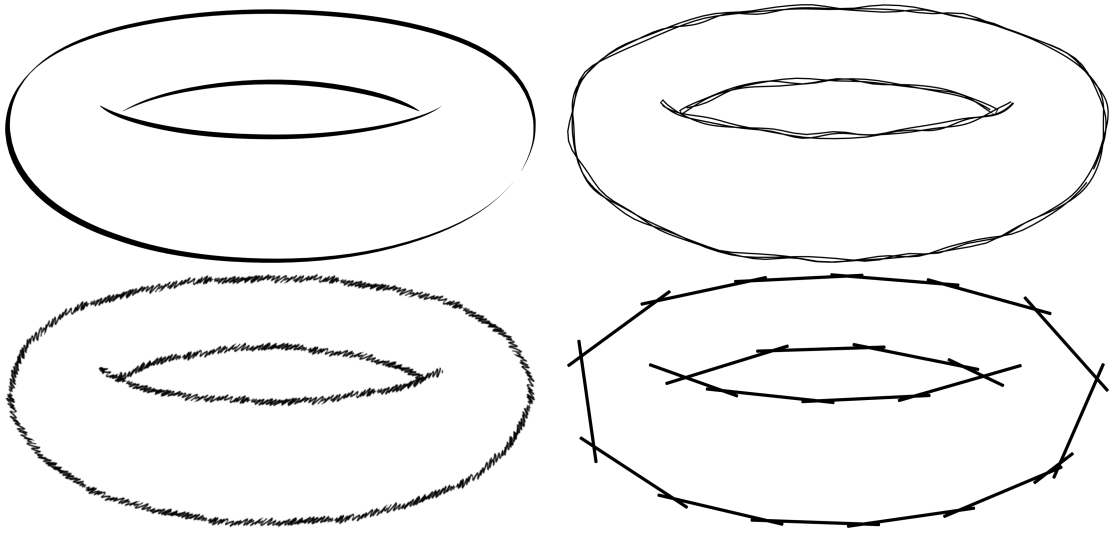


Figure 9.4: Stroke style — Various stroke styles applied to the contours of Figure 7.6b using Blender Freestyle. From left to right, top to bottom: calligraphy, sketchy overdraw, textured scribbles and guiding lines.

the radius of curvature is zero (Asente, 2010). The skeletal strokes can then be rendered as a series of triangular strips (Northrup and Markosian, 2000), or as quads with caps (McGuire and Hughes, 2004).

Stroke stylization may be defined by a texture map, such as a scanned pen stroke (Hsu and Lee, 1994), or procedurally by making attributes such as color, transparency, dashes (Dooley and Cohen, 1990; Elber, 1995b; Grabli et al., 2010; Northrup and Markosian, 2000) vary along the stroke (Figure 9.4). Artistic perturbations of the stroke by analytic (e.g., sine function) or noise functions and offsets (Markosian et al., 1997) can further add high-frequency variations to the stroke. To avoid tiling and stretching artifacts, texture synthesis can be used to generate arbitrary-length stroke textures (Bénard et al., 2010) and offsets (Hertzmann et al., 2002; Kalnins et al., 2002; Bénard et al., 2012; Lang and Alexa,

2015) from examples. Stroke styles can depend on the underlying source geometry, e.g., thicker strokes for nearer objects or less-curved objects (Goodwin et al., 2007), and stroke styles can also vary for different objects and materials. Stroke visibility may be "haloed" for greater clarity (Elber, 1995b). More sophisticated texture-based stroke models include "RealBrush", which uses multiple scanned paint strokes to model paint mixing (Lu et al., 2013) and "DecoBrush" (Lu et al., 2014), which uses procedurally-defined line art textures. These stroke textures may also be drawn directly in a WYSIWYG interface (Kalnins et al., 2002; Cardona and Saito, 2015, 2016). See the video accompanying the work of Kalnins et al. (2002) for a particularly inspiring interface.

An alternative approach is to reproduce the appearance of natural media, such as ink, paint, watercolor, or charcoal, using using physical models. These methods simulate the pigments deposited by a drawing tool (e.g., pen, pencil, brush) on a substrate (paper or canvas). For example, Curtis et al. (1997) reproduce watercolor strokes using fluid simulation to compute the motion of water and pigments deposited by a brush on a textured paper. Examples of similar simulations include oil paint (Baxter et al., 2004; Chen et al., 2015b) and graphite pencils (Sousa and Buchanan, 1999). The order in which strokes are drawn is usually important and an adequate blending model is thus required. If an accurate painting simulation is available, the Kubelka-Munk (Kubelka, 1948; Haase and Meyer, 1992) model of pigment layering can be used. Otherwise, simpler approximations can be used such as the OpenGL blending modes. To emulate thick media such as oil paint, a simple "replace" mode is usually sufficient. Substractive blending gives a decent approximation of the behavior of wet materials such as ink and watercolor. Finally the "minimum" blending mode can imitate dry media such as graphite and crayons.

9.3 Topological simplification

There are three reasons to perform a topological simplification step. First, computing smooth contours from meshes often creates overly complex contours, as discussed in Section 6.1. Second, even the correct contours may exhibit very tiny loops or other topological features that we would like to remove. Third, smoothing and filtering strokes is an important stylization step in creating artistic images. This includes removing strokes in overly dense regions. These different simplification and stylization steps are each, potentially, operations on the View Graph (or Planar Map), one cannot apply them to individual strokes in isolation.

Clean-up heuristics for mesh contours. When the underlying surface is smooth, each chain should be topologically equivalent to a line, which is essential for artifact-free stylized rendering. Unfortunately, due to numerical instabilities, topological errors might have been introduced during the contour extraction step, especially with polygonal mesh approximations. After projection onto the image plane, this leads to two main artifacts: overlapping contour edges and small "zig-zags". For example, when a mesh with low curvature is tangent to the view direction, multiple adjacent triangles may alternate between front- and back-facing orientation, producing a cluster of contour edges (Figure 4.4).

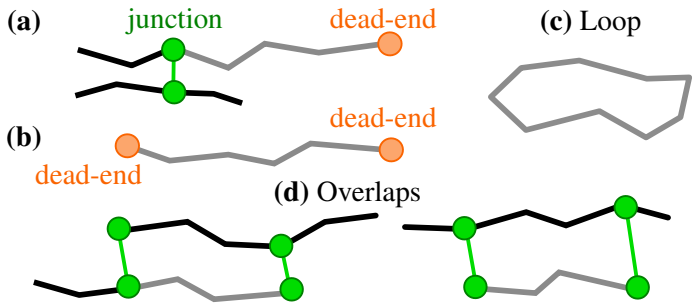


Figure 9.5: Topological simplification (Bénard et al., 2014) — Four cases are considered for simplification (candidate chain depicted in grey): **(a)** junction to dead-end connection; **(b)** dead-end to dead-end connection; **(c)** small closed loop; **(d)** small overlapping pieces of curve between two junctions.

Fast heuristics may be used to clean up these artifacts. For example, Northrup and Markosian (2000) describe a method that identifies image-space line segments that overlap and are nearly parallel; they eliminate redundant segments that are similar to another very nearby segment (parallel and close-by), and smaller than the other segment. This allows them to merge many small edges with complex topology into long, simple paths. In addition, Isenberg et al. (2002) perform a first simplification step in object-space to merge adjacent edges connecting at acute angles, and to remove spurious bifurcations produced by clusters of contour edges.

Foster et al. (2007) proposed an alternative approach based on multi-resolution filtering through reverse subdivision. Each contour chain is first decomposed into a coarse base path and a representation of its high-frequency details. It is then reconstructed to its original resolution with a scaled-down version of the details to remove errors.

Though these methods do not provide any topological guarantees, they can be fast and simple and produce appealing results.

Removing tiny details. Even if the input contour generators have correct topology, they may exhibit unappealing topological details such as tiny loops or breaks due to cusps (Figure 9.6a). To improve the appearance of the final rendered strokes, topological simplification can be applied to the View Graph chains (Figure 9.6b). Unlike the previous heuristics, this simplification is purely a stylistic control; one that depends on the scale of the objects as well as the rendering style, not an attempt to estimate and fix topology from noisy curves.

In particular, we proposed the following topological simplifications (Figure 9.5) (Bénard et al., 2014). First, we categorized View Graph vertices (i.e., singularities) by the number of visible curves they connect: *a dead-end* vertex is adjacent to a single visible curve (e.g., a visible curtain fold); a *connector* vertex is adjacent to two visible curves, and a *junction* vertex is adjacent to more than two vertices, i.e., bifurcations and image-space intersection vertices. Then, we defined a candidate chain for simplification as any connected sequence of visible curves that do not contain any junctions, but with image-space arc length less than

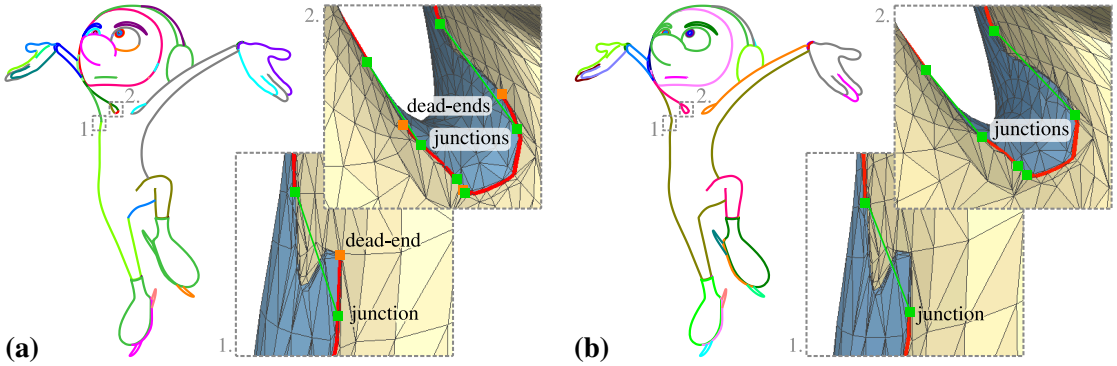


Figure 9.6: Result of topological simplification on Red (Bénard et al., 2014) — Closeup on Red's shoulder and armpit, with genuine cusps, **(a)** before and **(b)** after topological simplification. "Red" © Disney/Pixar

a user-specified threshold (between 10 and 20 pixels in our experiments). Eventually the algorithm marks as invisible any candidate chain that (a) connects a junction to a dead-end, (b) connects a dead-end to a dead-end, (c) connects a vertex to itself, or (d) is overlapped in 2D by another chain (Figure 9.5). This process is iterated until there are no more changes to be made.

Controlling image-based density. An artist drawing a small or distant object would draw only a few of its curves. However, directly computing all curves on a detailed surface from far away produces an overly-dense set of curves. Wilson and Ma (2004) propose a first method to omit excess curves from a drawing, based on the density of strokes in image space. Grabli et al. (2004) further distinguish between two kinds of line density measures: an *a priori* density and a *casual* density. The former estimates, at a given scale and for a given direction, the geometric complexity of the line drawing that would be created if all visible lines were drawn without stylization. The latter measures the actual visual complexity of the stylized drawing while it is rendered one curve at a time.

As noted by Winkenbach and Salesin (1994); Preim and Strothotte (1995), drawing simplification by line omission also requires ordering the curves by *relevance*, the least relevant ones being omitted first. Various definition of curve relevance have been proposed. For instance, to preserve curves that separate objects that are far apart in depth, the curve relevance can be defined as the maximum depth difference measured along its segments.

Scale-dependent contours. Another approach to curve simplification is to use a range of representations of the original surface, using a coarse version for rendering at a distance, and a detailed version for close-up viewing (Deussen and Strothotte, 2000; Jeong et al., 2005; Kirsanov et al., 2003; Ni et al., 2006). Image-space density can be used as a criteria for selecting the object level-of-details. An advantage of this approach is that it can ensure a coherent topology of the drawing; however, it may not adapt to naturally varying density as well object-based stylization, for example, for highly-foreshortened objects.

9.4 Object shading and texturing

In addition to line drawing, we usually wish to shade or texture the object. A thorough discussion of shading is beyond the scope of this tutorial. Generally, most methods compute shading independently from contours and other lines. In principle, doing so could create inconsistency between the line drawing and the shaded rendering, e.g., if the lines are smoothed or simplified. While this is not usually a problem, using a Planar Map to maintain a consistent representation can fix this issue (Eisemann et al., 2008; Winkenbach and Salesin, 1994).

Once a shaded image has been rendered from the 3D scene, any of the methods in the book of Rosin and Collomosse (2013) may be applied as post-processes to stylize images as well. Nevertheless, more dedicated shading primitives will allow to produce more stylized results.

Toon shading. A simple and popular shading algorithm is "toon shading". In toon shading, the shading is simply a function of the view vector and light direction: $\mathbf{n} \cdot \mathbf{l}$. In its simplest form, the user specifies two colors: a light color and a dark color. For shading, points where the dot product is below a threshold get rendered with the dark color; other points get rendered with the light color (Figure 1.1b). This generalizes contour rendering, which discretizes $\mathbf{n} \cdot \mathbf{v}$ into black at the contour and white everywhere else.

Several generalizations of toon shading have been developed, typically mapping $\mathbf{n}$ to color with a more complex function, e.g., (Lake et al., 2000; Sloan et al., 2001; Gooch, 1998; Barla et al., 2006; Mitchell et al., 2007; Eisemann et al., 2008; Vanderhaeghe et al., 2011).

Hatching and texturing. Surface hatching (Figure 1.6) often involves drawing hatching curves (Winkenbach and Salesin, 1994, 1996; Hertzmann and Zorin, 2000; Singh and Schaefer, 2010; Kalogerakis et al., 2012; Gerl and Isenberg, 2013). These are curves on the surface, and their visibility is computed together with the other curves, using the algorithms described in this tutorial.

A wide variety of algorithms for texturing have also been developed without drawing texture curves, instead using texture maps in some way, e.g., (Klein et al., 2000; Praun et al., 2001; Webb et al., 2002; Breslav et al., 2007).

Two methods for stylizing objects and animations by example using the Image Analogies framework of Hertzmann et al. (2001) have been developed (Bénard et al., 2013; Fišer et al., 2016).

9.5 Animation

To produce an animated line drawing, one can simply extract and stylize the contours at every frame independently. However, as first noted by Masuch et al. (1997), the coherence between frames largely depends on the chosen stroke style. A "calm" style with only small

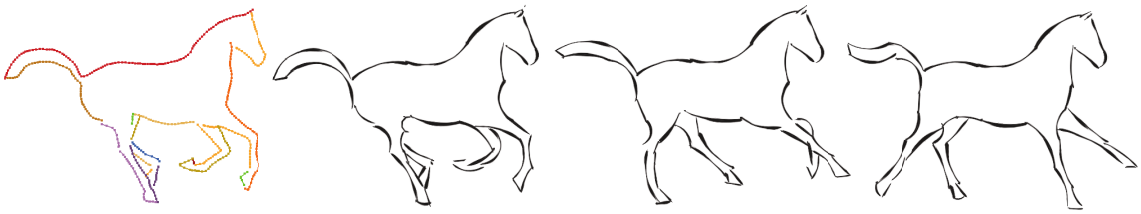


Figure 9.7: 2D curve tracking by active contours (Bénard et al., 2012) — The contour curves extracted from a galloping horse are tracked by active contours (left) ensuring a coherent parameterization of the strokes (right) during the animation.

deviations from the base path leads to a rather smooth animation, whereas a "wild" style with strong geometric distortions or using strong textures may lead to visual artifacts such as popping or sliding. This is a recurrent but still mostly open problem in non-photorealistic rendering (Bénard et al., 2011). In the following, we will summarize the main solutions to improve temporal coherence of line drawing animations.

2D curve tracking. The general objective is to establish correspondences between stokes of subsequent frames and derive from them a coherent space-time parameterization. View-independent lines, such as creases or ridges and valleys, that are fixed on the 3D surface can leverage the underlying surface parameterization to ensure such correspondences. In contrast, view-dependent lines such as contours move on the surface and their geometry and even topology change from frame to frame. Consequently most approaches directly compute correspondences in image-space. This is related to computer-assisted rotoscoping which aims to track edges in videos (Agarwala et al., 2004; O'Donovan and Hertzmann, 2012) with the benefit that perfect motion information can be computed from 3D animations. Disney "Paperman" (Whited et al., 2012) precisely follows such an approach to propagate hand-drawn strokes over CG renders. The main drawback of this system is that artists need to manually merge or split strokes when their topology should change to adapt to the animation.

The curve tracking method of Bénard et al. (2012) lifts this limitation. It uses active contours (a.k.a. snakes) that automatically update their position, arrangement and topology to match the contour animation (Figure 9.7). However, based on heuristics, this method cannot guarantee that the strokes are faithfully depicting the contours, especially at junctions.

For more robustness, Ben-Zvi et al. (2015) turn the problem of curve tracking into one of matching. Since matching full curves would be impractical due to topological changes, they seek to find a mapping between the vertices of the strokes in frame f_i and f_{i+1} (Figure 9.8). This mapping aims at minimizing the distance between matched vertices, after moving the points in f_i to f_{i+1} according to the animation motion field. The mapping should also maximize the number of matched vertices, favor one-to-one matches, and maintain the spatial ordering of the vertices on the strokes as much as possible. These objectives can be expressed as a constrained optimization problem on a bipartite graph whose nodes on

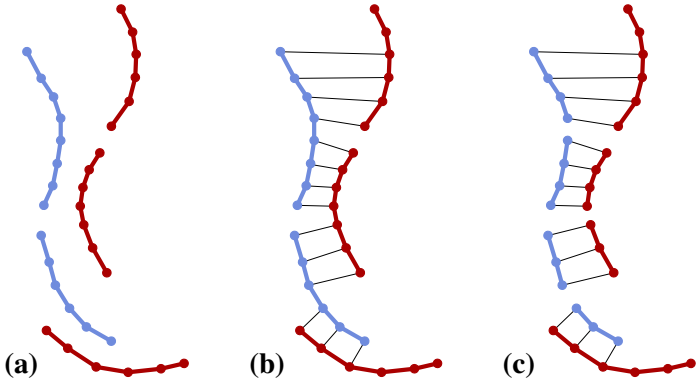


Figure 9.8: 2D curve matching (Ben-Zvi et al., 2015) — Starting from the strokes at frame f_i in blue and f_{i+1} in red **(a)**, point correspondences are computed **(b)** by a constrained optimization, from which consistent sub-strokes can be extracted **(c)** .

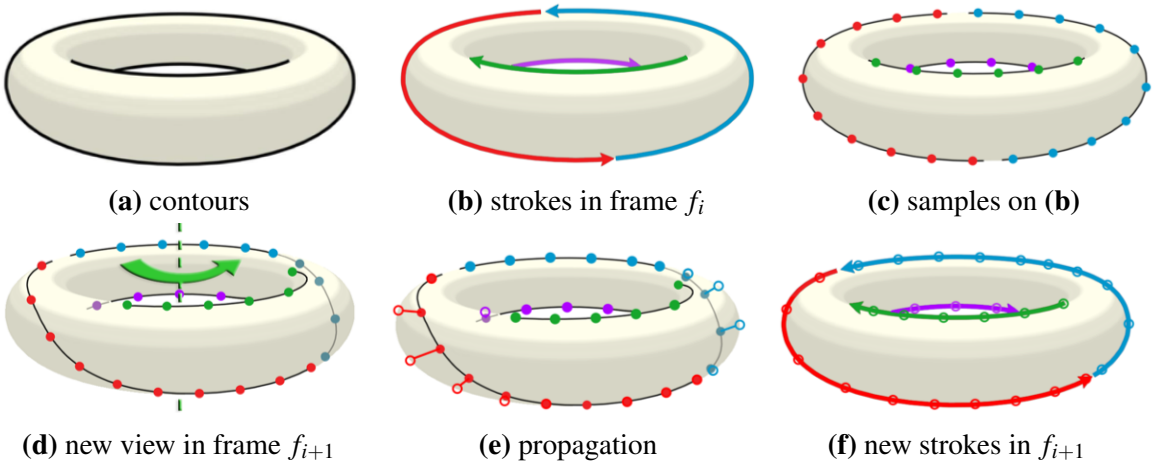


Figure 9.9: Parameterization propagation (Kalnins et al., 2003) — The strokes at frame f_i **(b)** are sampled uniformly along their arc-length **(c)** and propagated by reprojection in the new camera **(d)** and local search in image-space **(e)**. Coherently parameterized strokes are created by potentially splitting new contour curves and fitting a continuous function to the samples balancing uniformity in 2D with coherence in 3D **(f)**.

each side of the graph are the vertices in each frame and whose edges connect any pair of vertices from different frames. Coherent sub-strokes are eventually constructed from these point-wise correspondences by splitting inconsistent curves (Figure 9.8c). A major drawback of this method is the computation cost of the point-matching algorithm which does not scale well with the number of curves.

Parameterization propagation. Another solution to these topological issues is to propagate the parameterization of the strokes instead of their geometry. Building upon the work of Bourdev (1998), this is the key idea of Kalnins et al. (2003). They sample the parametrization of the strokes at frame f_i uniformly along their arc-length (Figure 9.9c), and

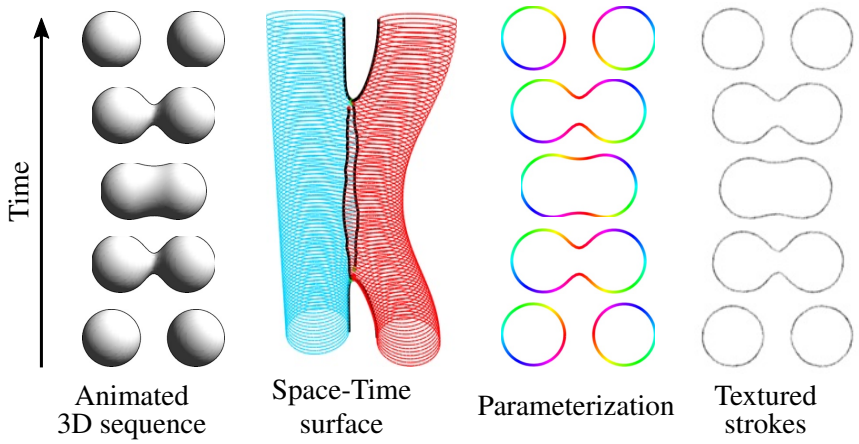


Figure 9.10: Space-time parameterization (Buchholz et al., 2011) — Temporally coherent textured strokes are computed by parameterizing the space-time surface swept by the contours over time.

reproject those samples in the camera of the next frame f_{i+1} following the 3D animation to approximate the contour motion (Figure 9.9d). Then, they locally search in image-space the location of the closest contour paths in the new view (Figure 9.9e). Samples from different brushes of frame f_i may end up on the same contour path in frame f_{i+1}. In such a case, the contour path needs to be split into multiple strokes with consistent parameterization samples. Eventually, each stroke parameterization is computed by optimizing an energy function that balances the competing goals of uniform image-space arc-length parameterization and coherence on the object surface. This method ensures a temporally coherent parameterization for contours with simple topology at interactive framerates. However, since more complex objects, such as the Stanford Bunny, generate contours made of many tiny fragments, topological simplification (Section 9.3) needs to be applied first, otherwise the strokes may get increasingly fragmented over time.

Space-time parameterization. The above methods consider only two animation frames at a time. Buchholz et al. (2011) proposed considering the entire animation as a whole, building a space-time parameterization of the strokes over time (Figure 9.10). To do so, they build the space-time surface swept by the contours over time, taking into account merging and splitting events. This allows minimizing distortions and "popping" artifacts over the whole sequence, rather than greedily processing each frame one-at-a-time. This comes however at the price of expensive computations (up to several minutes for few seconds of animation).

Motion of contours across time. Some theoretical analysis of how contours evolve over time is provided by Cipolla and Giblin (2000); Plantinga and Vegter (2006). Plantinga and Vegter (2006) additionally provide a fast algorithm for tracking contours of implicit surfaces over time, with guaranteed topological correctness, under orthographic projection.

10 CONCLUSION

The techniques we have described here provide an account for how to make line drawings from 3D models. In organizing the information from the past few decades of research on this topic, we hope that this material will be useful for future practitioners and researchers.

The methods described here present a range of design choices. On one extreme, hardware rendering methods allow real-time performance and relatively simple implementation, but only a narrow range of rendering styles. Interpolated contours are much more flexible and allow for many more rendering styles, though with some implementation effort, and some potential artifacts appearing on smooth surfaces. In the other extreme, methods that attempt to correctly compute all curves for smooth surfaces are currently the most complex; providing full correctness guarantees remains a research problem. Within this range of options, there are more design choices to be made, such as which visibility tests to use besides ray tests, which strategies to use to propagate visibility, what numerical robustness strategies to try. These choices make tradeoffs between stylistic control, accuracy, efficiency, and complexity of implementation. Our community does not yet have the experience of building real systems that would allow us to make recommendations about many of these choices. However, at a high level, knowing one's requirements for stylistic variation, real-time performance, and accuracy can guide one to one of the three main approaches listed above.

10.1 Open research problems

The work presented here has been developed for applications in entertainment, art, and scientific visualization. There are several games that have used hardware line-drawing methods, and 3D methods have shown up in a few films here and there, such as in Disney's Paperman, and the Freestyle NPR line drawing algorithms have been incorporated in the free Blender package. Still, many of the most sophisticated methods here have not made it into commercial use.

These research areas were very active in the 90's and 2000's, and now there is little effort. This is particularly evident at SIGGRAPH, the flagship venue for this research; now most research, when it appears, is at more specialized, less-impactful venues. There are many possible reasons for this stagnation. We believe that it does not reflect a lack of interest in these problems, but, rather, the difficulty for many researchers outside the area in identifying

known research projects. It should be obvious from this tutorial that there are some clear, open research problems that are purely geometric in nature.

For professional stylization applications (e.g., films), one would ideally like to have a space-time planar map of a scene, including space-time correspondences and complete curve topology, and no existing system for this has even been attempted, to our knowledge. The engineering effort involved may be well beyond what is achievable by a single graduate student working alone, without truly heroic effort and understanding of the geometric, numerical, and artistic considerations involved.

Once developed, such a system would create a variety of new engineering, authoring, and workflow challenges and opportunities.

Since these problems first arose, there has been tremendous progress in other areas of geometric modeling and simulation. The fundamental difficulty for correct curve topology is that a single incorrect visibility test at a seemingly-insignificant little triangle can cause enormous visibility errors. This parallels problems that can occur in other areas of modeling and simulation. For example, when simulating a hanging piece of cloth, a single little missed collision can lead to massive interpenetrations that ruin the simulation: a single non-robust test causes topological catastrophe. The geometric modeling and simulation communities have developed robust geometric tools and techniques to prevent these problems. It may now be time to revisit the line drawing problem with this new knowledge in hand.

A separate problem is to build artist-friendly tools for authoring artistic styles. There have been various approaches explored in the literature, such as rotoscoping (Kalnins et al., 2002; Sabiston, 2001; Whited et al., 2012; Cardona and Saito, 2015) and procedural authoring (Grabli et al., 2010).

Machine learning and example-based rendering could provide a way to author styles. So far, there has been very little effort combining machine learning and 3D NPR; exceptions include (Kalogerakis et al., 2012; Bénard et al., 2013; Fišer et al., 2016).

Whatever the future work, the methods described in this tutorial provide the first part of the story, but the rest is yet to be written.

10.2 Acknowledgements

We thank Doug DeCarlo for comments on a draft of this tutorial, and a reviewer for extremely detailed comments.

A

Fundamentals of Differential Geometry

This chapter presents the fundamentals of differential geometry that are useful to define smooth surface contours. It is based on the books and courses of Cipolla and Giblin (2000); Rusinkiewicz et al. (2008); Crane et al. (2013).

A.1 Geometry of surfaces

The geometry of a 3D surface can be described using a map $f : M \subset \mathbb{R}^2 \to \mathbb{R}^3$ from a region M in the Euclidean plane $\mathbb{R}^2$ to a subset $f(M)$ of $\mathbb{R}^3$. It is called a parametric surface if f is an *immersion*, that is its partial derivatives $\frac{\partial f}{\partial u}$ and $\frac{\partial f}{\partial v}$ are injective at each point of M. In this case, f defines an *immersed surface*; a surface where, for every point $\mathbf{u}$ of M, a definite 2-dimensional tangent plane is associated at $\mathbf{p} = f(\mathbf{u})$, or, simply, $\mathbf{p}(\mathbf{u})$ (Figure A.1).

A vector $\mathbf{n} \in \mathbb{R}^3$ is *normal* to the tangent plane at $\mathbf{p}$ if, for all tangent vectors $\mathbf{t}$ at $\mathbf{p}$, $\mathbf{n} \cdot \mathbf{t} = 0$, where $\cdot$ is the canonical Euclidean dot product on $\mathbb{R}^3$. Since the tangent plane at $\mathbf{p}$ can be defined as a linear combination of the partial derivatives of the immersion f, the *unit*

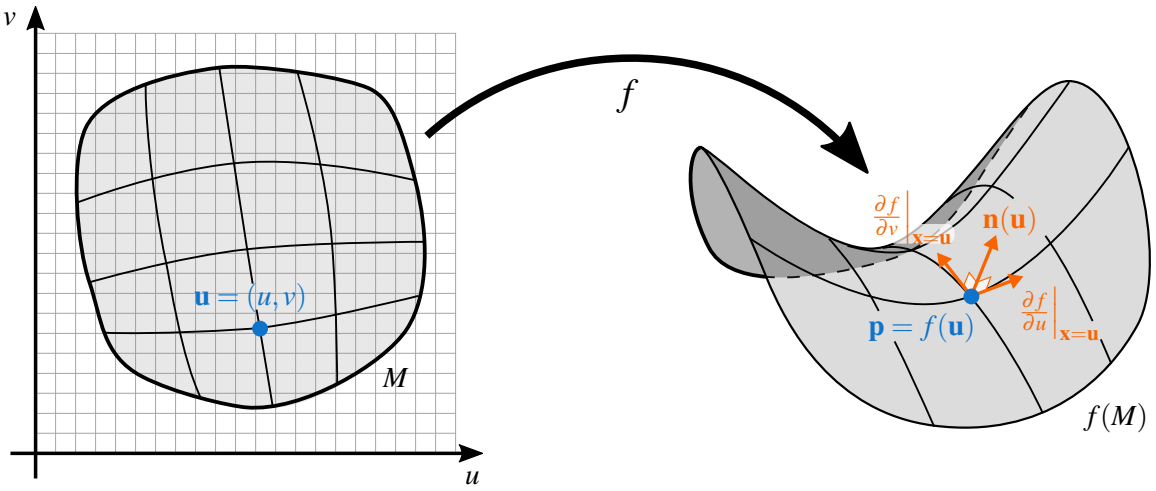


Figure A.1: Parametric surface — The immersive map $f : M \subset \mathbb{R}^2 \to \mathbb{R}^3$ associates to each parametric location $\mathbf{u}$ a unique 3D point $\mathbf{p}$; the normal $\mathbf{n}$ at that point is defined as the cross-product of the partial derivatives of the parameterization.

surface normal $\mathbf{n}$ at $\mathbf{p}$ is defined as:

$$\mathbf{n}(\mathbf{u}) = \frac{\left.\frac{\partial f}{\partial u}\right|_{\mathbf{x}=\mathbf{u}} \times \left.\frac{\partial f}{\partial v}\right|_{\mathbf{x}=\mathbf{u}}}{\left\| \left.\frac{\partial f}{\partial u}\right|_{\mathbf{x}=\mathbf{u}} \times \left.\frac{\partial f}{\partial v}\right|_{\mathbf{x}=\mathbf{u}} \right\|}.$$

Note that interchanging the two partial derivatives takes $\mathbf{n}$ to $-\mathbf{n}$. If we can choose a consistent direction for $\mathbf{n}$ for all points $\mathbf{p}$, M is *orientable*. For orientable surfaces, $\mathbf{n}$ can be seen as a continuous map, called the *Gauss map*, which associates each point of M with its unit normal, viewed as a point on the unit sphere S^2.

A.2 Curvature

Informally, the curvature describes how much a surface bends at a certain point and in a particular direction. For instance, an (infinite) cylinder curves around in a circle along one direction, and is completely flat along another direction. It is common to treat surface curvature in terms of 3D curves contained in the surface. We thus first need to define the curvature of a 3D curve.

Curvature of a curve. Let $c : I \subset \mathbb{R} \to \mathbb{R}^3$ be a 3-dimensional parametric curve with *unit speed*, i.e.,with *arc-length* or *natural* parameterization: $\left\| \frac{dc}{dt} \right\| = 1$. The curvature of c is measured by the rate at which the *unit* tangent vector changes as we move along c. This change is split into two pieces: the unit vector $\mathbf{n}$, called the *principal normal*, which describes the direction of change, and a scalar $\kappa \in \mathbb{R}$, called the *curvature*, which expresses the magnitude of change:

$$\frac{d^2c}{dt^2} = \mathbf{t}' = -\kappa\mathbf{n}.$$

Assuming that κ is never zero, the plane spanned by $\mathbf{t}$ and $\mathbf{n}$ is the *osculating plane*. The vector $\mathbf{b}$ orthogonal to the osculating plane is called the *binormal* of the curve: $\mathbf{b} = \mathbf{t} \times \mathbf{n}$. The orthonormal coordinate frame made of $\mathbf{t}, \mathbf{n}, \mathbf{b}$ is called the *Frenet frame* (Figure A.2).

For any point $\mathbf{p} = c(t)$, the circle S in the osculating plane with centre $\mathbf{p} + \mathbf{n}/\kappa$ is called the *osculating circle* of c at $\mathbf{p}$. This circle best approximates c at $\mathbf{p}$, meaning that it has the same tangent direction $\mathbf{t}$ and curvature vector $\kappa\mathbf{n}$, or that their first and second derivatives agree. It corresponds to the circle passing through $\mathbf{p}$ and two infinitely close points on c, one approaching from the left and one from the right of $\mathbf{p}$. The radius and center of the osculating circle are often referred to as the *radius of curvature* and *center of curvature*, respectively.

The *torsion* τ of c measures the tendency of the curve to leave its osculating plane, i.e., the way the normal and binormal twist around the curve. The *Frenet-Serret formula* describes how the $\mathbf{t}, \mathbf{n}, \mathbf{b}$ frame changes along the curve:

$$\mathbf{t}' = -\kappa\mathbf{n}, \qquad \mathbf{n}' = \kappa\mathbf{t} - \tau\mathbf{b}, \qquad \mathbf{b}' = \tau\mathbf{n}.$$

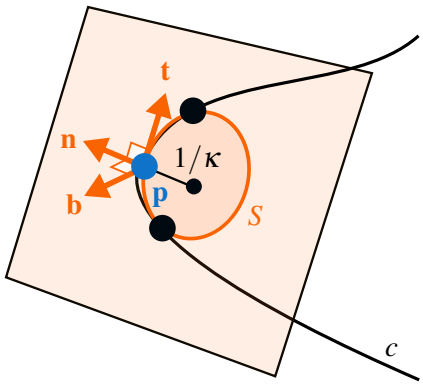


Figure A.2: Curvature of a curve — The curvature κ of the 3D parametric curve c at $\mathbf{p} = c(t)$ corresponds to the inverse of the radius of the osculating circle S in the osculating plane (in gray) defined by the tangent $\mathbf{t}$ and principal normal $\mathbf{n}$.

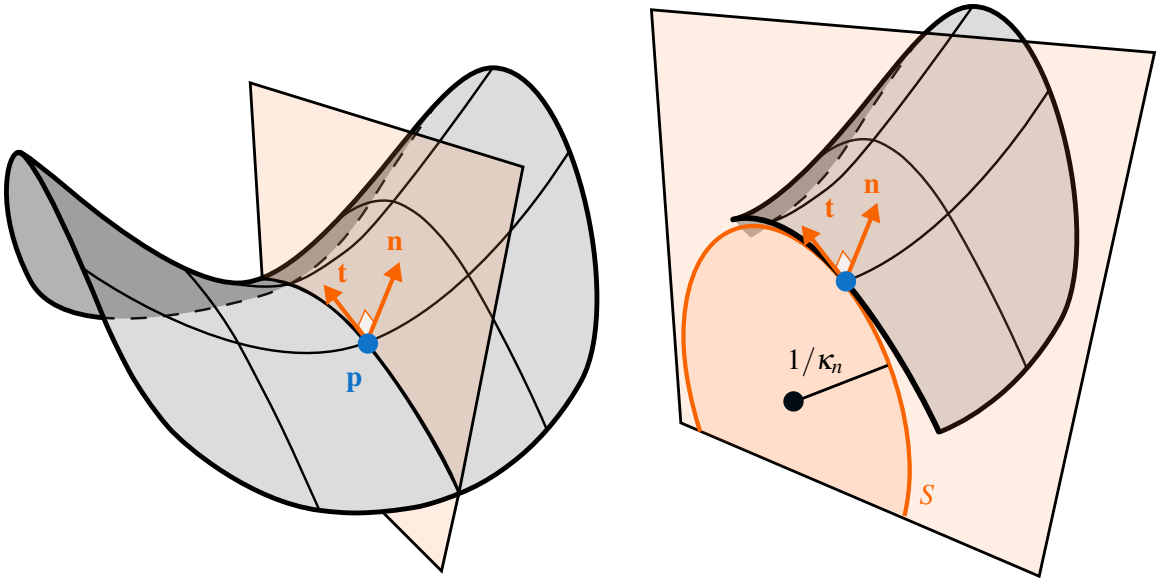


Figure A.3: Normal curvature — The curvature κ_n of the curve at the intersection of the surface $f(M)$ and the plane spanned by the normal $\mathbf{n}$ and tangent direction $\mathbf{t}$ is the normal curvature of M at $\mathbf{p}$ in the direction $\mathbf{t}$.

For our purpose, the important formula is the second one, which shows that we can get the curvature by extracting the tangential part of $\mathbf{n}'$.

Normal curvature of a surface. Going back to surfaces, consider the plane containing a given point $\mathbf{p}$ on the surface $f(M)$, a vector $\mathbf{t}$ in the tangent plane at that point and the associated normal $\mathbf{n}$. This plane intersects the surface in a curve, and the curvature κ_n of this curve is called the *normal (or sectional) curvature* in the direction $\mathbf{t}$ (Figure A.3). Using the Frenet-Serret formula, we can get the normal curvature along $\mathbf{t}$ by extracting the tangential

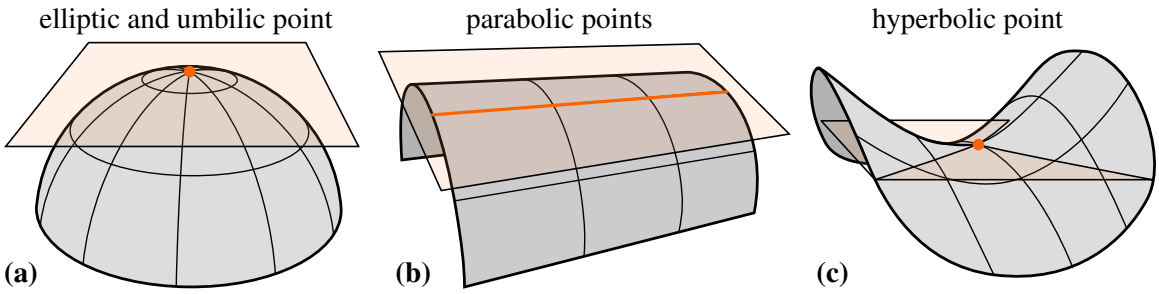


Figure A.4: Surface categorization based on Gaussian curvature.

part of $\mathbf{n}'$:

$$\kappa_n(\mathbf{t}) = \frac{\mathbf{t} \cdot \mathbf{n}'}{\|\mathbf{t}\|^2}.$$

This means that the normal curvature is a measure of how much the normal changes in the direction $\mathbf{t}$. Note that it is signed, meaning that the surface can bend toward or away from the normal, but it is not affected by the length of $\mathbf{t}$. Since $\mathbf{n}$ is a unit vector, its derivative $\mathbf{n}'$ is perpendicular to $\mathbf{n}$, hence in the tangent plane of $\mathbf{p}$. $\mathbf{n}'$ is also called the *shape operator* $S(\mathbf{t})$.

Principal, Gaussian and Mean curvature. For a given point $\mathbf{p}$, the unit vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ along which the normal curvature is maximal and minimal, respectively, are called the *principle directions* at $\mathbf{p}$; the associated curvature values κ_1 and κ_2 are called the *principal curvatures*. If $\kappa_1 = \kappa_2$, every direction is principal and the point is called an *umbilic* (Figure A.4a). Otherwise, there are two orthogonal principle directions (i.e., $\mathbf{e}_1 \cdot \mathbf{e}_2 = 0$). Principal directions and curvatures respectively corresponds to eigenvectors and eigenvalues of the shape operator:

$$S(\mathbf{e_i}) = \kappa_i \mathbf{e_i}.$$

The *Gaussian* curvature K is equal to the product of the principal curvatures:

$$K = \kappa_1 \kappa_2,$$

and the mean curvature H is their average:

$$H = \frac{\kappa_1 + \kappa_2}{2}.$$

Based on the sign of its Gaussian curvature, a surface point is called (Figure A.4):

- *elliptic* if $K > 0$, i.e., κ_1 and κ_2 have the same sign;
- *parabolic* if $K = 0 \Rightarrow \kappa_1 = 0$ or $\kappa_2 = 0$;
- *hyperbolic* if $K < 0$, i.e., κ_1 and κ_2 have opposite signs.

Surfaces with zero Gaussian curvature are called *developable surfaces*, because they can be flattened out into a plane without distortions. For example, a cylinder always has one of its principal curvature equal to zero. Surfaces with zero mean curvature are called *minimal surfaces* because they locally minimize surface area. Since $\kappa_1 = -\kappa_2$ on minimal surfaces, they tend to look like saddles, which is also a good example of surfaces with negative Gaussian surfaces. On the other hand, surfaces with positive Gaussian curvature tend to look like hemispheres.

The fundamental forms. Even though they do not introduce new geometric ideas, the fundamental forms are important for historical reasons. The first fundamental form $\mathbf{I}$ corresponds to the metric induced by the map f, which measures the inner product between any two vectors $\mathbf{x}$, $\mathbf{y}$ in the tangent plane at $\mathbf{p}$:

$$\mathbf{I}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}.$$

The second fundamental form $\mathbf{II}$ at $\mathbf{p}$ is a symmetric bilinear form specified by:

$$\mathbf{II}(\mathbf{x}, \mathbf{y}) = S(\mathbf{x}) \cdot \mathbf{y} = S(\mathbf{y}) \cdot \mathbf{x}.$$

With those notations, the normal curvature in the tangent direction $\mathbf{t}$ can be re-written as:

$$\kappa_n = \frac{\mathbf{II}(\mathbf{t}, \mathbf{t})}{\mathbf{I}(\mathbf{t}, \mathbf{t})}.$$

Principal coordinates. Using the principal directions $\mathbf{e}_1, \mathbf{e}_2$ as a local basis for the tangent plane at $\mathbf{p}$ leads to the *principal coordinates*. When the vectors $\mathbf{x}$ and $\mathbf{y}$ are expressed in principal coordinates, the second fundamental form corresponds to the following diagonal matrix:

$$\mathbf{II}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \begin{bmatrix} \kappa_1 & 0 \\ 0 & \kappa_2 \end{bmatrix} \mathbf{y}.$$

This leads to Euler formula stating that the normal curvature in the direction $[\cos\theta, \sin\theta]^\top$ expressed in principal coordinates, where θ is the angle measured between this direction and $\mathbf{e}_1$, is:

$$\kappa_n(\theta) = \kappa_1 \cos^2\theta + \kappa_2 \sin^2\theta.$$

B

Convex and Concave Contours

We now show that concave contour edges cannot be visible.

Any triangle lies in a supporting plane that cuts space into two half-spaces. The surface normal points to one of these half-spaces. We can say that a shape is **in front of** the face when it is in the half-space that the normal points to. For example, a face is **front-facing** when the camera **c** is on the front side of the face. Likewise, a face is back-facing when the camera is **in back of the face**.

Definition B.0.1 (Concave/convex edge)**.** *A mesh edge on an orientable mesh is* concave *if each triangle is in front of the other. (That is, triangle A is in front of triangle B and vice-versa.) Otherwise, if they are each in back of the other, the face is* convex.

These cases are visualized in Figure B.1. It is easier to think about convexity in terms of the angle θ between the two oriented faces: the edge is convex if $\theta < \pi$, and concave otherwise. (If $\theta = \pi$, than the edge can never be a contour.) However, computing θ is somewhat tricky.

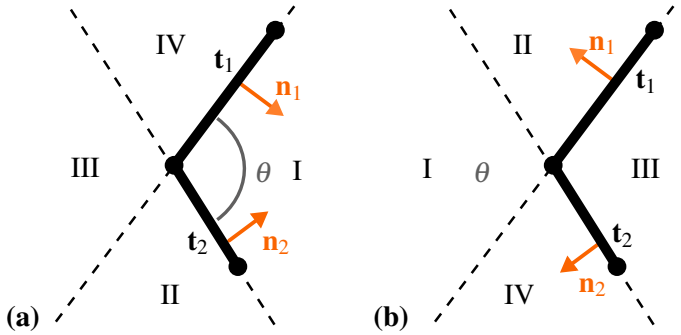


Figure B.1: Concave/convex edge — Cross-section plane for a mesh edge between triangles $\mathbf{t}_1$ and $\mathbf{t}_2$, with normals $\mathbf{n}_1$ and $\mathbf{n}_2$. The cross-section is some 3D plane perpendicular to the edge between the two triangles. The two faces divide space in four quadrants: I,II,III, and IV. For example, I is the set of points in front of both faces, and II is in front of $\mathbf{t}_1$ and behind $\mathbf{t}_2$. **(a)** Concave case: $\theta < \pi$. Each face is in front of the other. In this case, the edge cannot be a visible contour. **(b)** Convex case: $\theta > \pi$. Each face is behind the other.

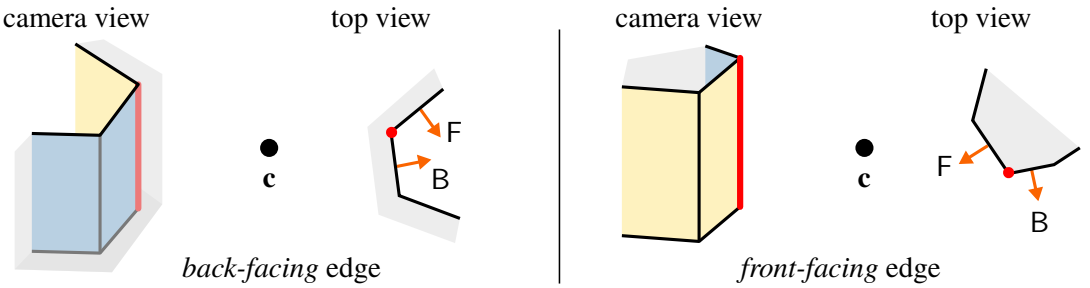


Figure B.2: Front-facing/back-facing edges — A contour generator edge is *back-facing*, resp. *front-facing*, when its adjacent face nearest to the camera position **c** is back-facing (B), resp. front-facing (F). The "interior" of the mesh (assumed closed) is depicted in grey for illustration purpose. Front-facing contour edges are always convex edges on the surface, and back-facing contours are always concave.

Intuitively, concave edges should not be visible when they are contours, because, from the camera's point-of-view, they are hidden inside the surface.

Theorem B.0.1. *On an orientable mesh where back-faces are never visible, a contour on a concave edge is never visible.*

Proof. The two faces on the edge divide 3D space into four quadrants (Figure B.1a) where the viewpoint **c** can lie. For the edge to be a contour, the viewpoint must be in either quadrant II or IV; otherwise, both faces are either front-facing or back-facing. If the camera is in quadrant II, triangle $\mathbf{t}_2$ is back-facing, and thus must be invisible. Moreover, $\mathbf{t}_2$ occludes $\mathbf{t}_1$, at least in the vicinity of the edge. Hence, the edge is invisible. The same reasoning directly applies to case IV. □

Furthermore, a contour on a convex edge is, locally, visible. While a convex contour could be occluded by some other surface far away, it lies on a front-face and so may be visible and is not locally occluded.

Front-facing and back-facing edges. Markosian et al. (1997) first introduced a version of these ideas called *front-facing edges* and *back-facing edges*. They defined a contour edge as *front-facing* if its adjacent face nearest the camera is front-facing, otherwise it is back-facing (Figure B.2). They mention in a footnote that they use convex/concavity instead of this definition.

Unfortunately, this definition does not always work. The distance between the camera and a face is the distance to the nearest point on the face. Figure B.3 shows a counterexample where their definition fails. In many situations, however, their method would be equivalent to computing convex/concave.

That said, the definition only fails in extreme cases, and the definition is useful for intuition when looking at diagrams.

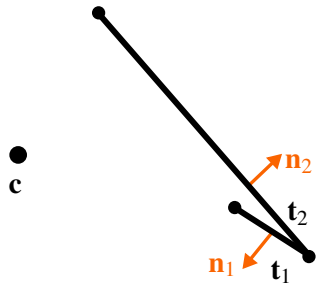


Figure B.3: Counterexample for Markosian's definition of front/back-facing edges — In this example, the face nearest to the camera is $\mathbf{t}_2$, which is back-facing. Therefore the contour edge between the triangles would be marked as back-facing and thus invisible. In fact, $\mathbf{t}_1$ occludes $\mathbf{t}_2$, even though $\mathbf{t}_2$ is nearer to the camera. This is a convex edge, and thus potentially visible.

C

ACCURATE NUMERICAL COMPUTATION

The visibility computations determine the topology of the View Graph, and minor errors can cause major topological errors. Hence, it is important to use robust geometric computations wherever possible.

One simple approach to improving precision is to use high-precision arithmetic (e.g., `float128`), infinite precision arithmetic, and/or rational arithmetic (since rational numbers are closed under perspective and intersection operations).

C.1 Logical intersections

During visibility, there are many different intersection computations. Whenever possible, these intersections should be done using pointers and logic, rather than numerics. In principle, all intersections could be detected in the image-space intersection step. However, numerical error could cause intersections to be missed this way. For example, a contour edge intersecting a boundary edges can be detected simply by finding edges that share a vertex. Smooth curves, on the other hand, lie within mesh faces; their end-points lie with mesh edges. Their intersections with mesh boundaries amount to determining if they share the same mesh edge.

Keeping track of pointers is also necessary for avoiding spurious intersections. For example, when performing a ray test on a contour edge, it is important that neither of the adjacent triangles unintentionally "occludes" the edge.

C.2 Orientation test

The orientation test is a useful building block of computational geometry. Many of the tests described here can be implemented in terms of the orientation test, and robust libraries exist for this test, such as the predicates of Shewchuk (1997)[1].

In 3D, the orientation test determines whether a point **d** lies to the left of, to the right of, or on the oriented plane defined by three other points **a**, **b** and **c**, appearing in counter-clockwise order when viewed from above the plane (Figure C.1). Two points are thus on the

[1] `http://www.cs.cmu.edu/~quake/robust.html`

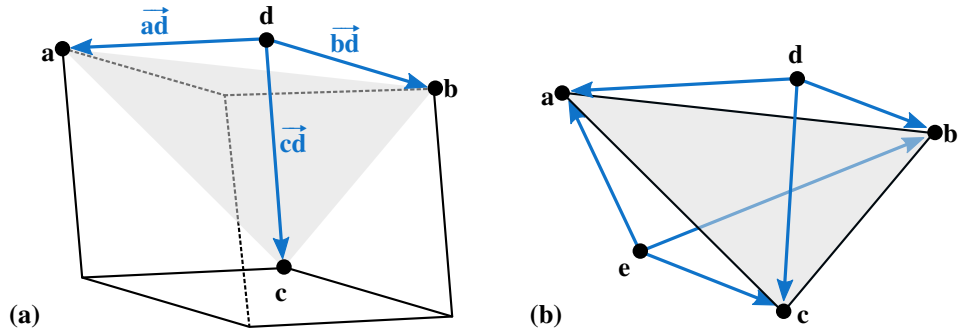


Figure C.1: 3D orientation and sidedness tests — The 3D orientation test **(a)** determines whether the point **d** is above, below or on the supporting plane of the oriented triangle $\triangle$**abc**, i.e., whether the signed volume of the parallelepiped spanned by the vectors $\overrightarrow{\mathbf{ad}}$, $\overrightarrow{\mathbf{bd}}$ and $\overrightarrow{\mathbf{cd}}$ is positive, negative or zero. Two points **d** and **e** are then on the opposite sides of the triangle $\triangle$**abc** if the their orientation tests have opposite signs **(b)**.

opposite sides of a triangular face if the results of their orientation tests with the triangle's supporting plane have opposite signs.

In any dimension, the orientation test can be implemented as a matrix determinant. In 3D, this is equivalent to the signed volume of the parallelepiped spanned by the vectors $\overrightarrow{\mathbf{ad}} = (\mathbf{a}-\mathbf{d})$, $\overrightarrow{\mathbf{bd}} = (\mathbf{b}-\mathbf{d})$, and $\overrightarrow{\mathbf{cd}} = (\mathbf{c}-\mathbf{d})$. That is, the side of the triangle $\triangle$**abc** that **d** lies on is determined by the sign of the scalar triple product:

$$\begin{aligned}
\textsc{Orient3D}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) &= \overrightarrow{\mathbf{ad}} \cdot (\overrightarrow{\mathbf{bd}} \times \overrightarrow{\mathbf{cd}}) \\
&= \det(\overrightarrow{\mathbf{ad}}, \overrightarrow{\mathbf{bd}}, \overrightarrow{\mathbf{cd}}) \\
&= \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \\
&= \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}
\end{aligned}$$

Highly-accurate libraries exist for these routines. The second determinant formula shows that swapping input parameters will flip the sign of the output, e.g., $\textsc{Orient3D}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) = -\textsc{Orient3D}(\mathbf{b},\mathbf{a},\mathbf{c},\mathbf{d})$.

The main use for orientation tests is to determine whether two points are on the same or the opposite side of a triangle. For this, we can define a function $\textsc{SameSide}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{e})$ that returns **true** if **d** and **e** are on the same side of $\triangle$**abc**, that is:

$$\begin{aligned}
&\textsc{SameSide}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{e}) \\
&= (\textsc{Orient3D}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) > 0) == (\textsc{Orient3D}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{e}) > 0)
\end{aligned}$$

In some cases, we also need to evaluate whether a point is on the front-facing side of a triangle, or the back-facing side. For this, we can define a function FRONTSIDE$(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d})$ that returns a positive value if $\mathbf{d}$ is on the front side of $\triangle\mathbf{abc}$. The surface normal (unnormalized), is given by $\mathbf{n} = \overrightarrow{\mathbf{ba}} \times \overrightarrow{\mathbf{ca}}$, and so:

$$\begin{aligned}
\text{FRONTSIDE}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) &= \overrightarrow{\mathbf{da}} \cdot \mathbf{n} \\
&= \overrightarrow{\mathbf{da}} \cdot (\overrightarrow{\mathbf{ba}} \times \overrightarrow{\mathbf{ca}}) \\
&= \det(\overrightarrow{\mathbf{da}}, \overrightarrow{\mathbf{ba}}, \overrightarrow{\mathbf{ca}}) \\
&= \text{ORIENT3D}(\mathbf{d},\mathbf{b},\mathbf{c},\mathbf{a}) \\
&= -\text{ORIENT3D}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d})
\end{aligned}$$

C.2.1 Applications of the orientation test

Several of the tests used here can be implemented with the orientation test:

Front-facing. Given camera position $\mathbf{c}$, the face $\triangle\mathbf{abd}$ is front-facing if FRONTSIDE$(\mathbf{a},\mathbf{b},\mathbf{d},\mathbf{c}) > 0$.

Concave edge. (Section 4.2). A mesh edge with vertices $\mathbf{a}$ and $\mathbf{b}$ with triangles $\triangle\mathbf{abd}$ and $\triangle\mathbf{bae}$ is concave if: FRONTSIDE$(\mathbf{a},\mathbf{b},\mathbf{d},\mathbf{e}) > 0$, or, equivalently, if FRONTSIDE$(\mathbf{b},\mathbf{a},\mathbf{e},\mathbf{d}) > 0$.

Image-space intersection. Image-space intersections are normally detected by the sweep-line algorithm, for efficiency. However, they can also be expressed in terms of 2D orientation tests. The 2D orientation test SAMESIDE2D is a 2D version of the one described above. Using image-space coordinates, two line segments $\mathbf{ab}$ and $\mathbf{de}$ intersect if **not** SAMESIDE2D$(\mathbf{d},\mathbf{e},\mathbf{a},\mathbf{b})$ **and not** SAMESIDE2D$(\mathbf{a},\mathbf{b},\mathbf{d},\mathbf{e})$.

Alternately, one can test for intersection without computing the 2D projection at all. The problem is equivalent to testing whether the 3D triangles $\triangle\mathbf{abc}$ and $\triangle\mathbf{cde}$ intersect, where $\mathbf{c}$ is the camera position, which amounts to two 3D SAMESIDE tests.

Overlap test for image-space intersection. (Section 4.3). Suppose we have determined that 3D segments $\mathbf{ab}$ and $\mathbf{de}$ intersect in image space, viewed from camera $\mathbf{c}$. Suppose line segment $\mathbf{ab}$ is a contour or boundary, and it is in front of the other segment at this point. Let $\triangle\mathbf{abf}$ be an adjacent triangle on the mesh. We need to determine which side of $\mathbf{de}$ is occluded by the surface. This amounts to determining whether SAMESIDE$(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{f})$ is true.

Boundary curtain fold. (Section 4.3). Curtain folds can occur on mesh boundaries (Figure 4.2 ④). Since there is no analogue of concave/convex edges for mesh boundaries, a different test is required. It consists in checking whether faces adjacent to a boundary vertex overlap in image-space, which can be computed by checking whether any non-adjacent

face of the one-ring neighborhood of the boundary vertex (brown triangles in Figure 4.3) occludes the boundary edge which is the farthest from the camera.

This can be reduced to three clipping tests (Figure 4.3); the edge **pe** is occluded by the face $\triangle$**pqr** if and only if: (1) **c** and **e** are on opposite sides of this triangle, i.e., SAMESIDE$(\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{c},\mathbf{e})$ is false, (2) **e** and **r** are on the same side of the triangle $\triangle$**cpq**, i.e., SAMESIDE$(\mathbf{c},\mathbf{p},\mathbf{q},\mathbf{e},\mathbf{r})$ is true, (3) **e** and **q** are on the same side of the triangle $\triangle$**cpr**, i.e., SAMESIDE$(\mathbf{c},\mathbf{p},\mathbf{r},\mathbf{e},\mathbf{q})$ is true. (This same test can also be used to detect curtain folds on contours, but it a simpler test exists for that case.)

It is possible, though unlikely, that a vertex has two boundary edges emerging from it, and both are locally-occluded. In this case, the above test produces a spurious curtain fold. These cases can be detected by performing the above local-overlap test on both boundary edges. This additional test is optional, since spurious curtain folds should not affect the final visibility results.

BIBLIOGRAPHY

A. Agarwala, A. Hertzmann, D. H. Salesin, and S. M. Seitz. Keyframe-based tracking for rotoscoping and animation. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 584–591. ACM, 2004. doi:10.1145/1186562.1015764.

A. Appel. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of the 1967 22Nd National Conference*, ACM '67, pages 387–393. ACM, 1967. doi:10.1145/800196.806007.

R. Arnheim. *Art and Visual Perception: A Psychology of the Creative Eye*. University of California, 2nd edition, 1974.

P. J. Asente. Folding avoidance in skeletal strokes. In *Proceedings of the Seventh Sketch-Based Interfaces and Modeling Symposium*, SBIM '10, pages 33–40. Eurographics Association, 2010. ISBN 978-3-905674-25-5.

P. Barla, J. Thollot, and L. Markosian. X-toon: An extended toon shader. In *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '06, pages 127–132, New York, NY, USA, 2006. ACM. ISBN 1-59593-357-3. doi:10.1145/1124728.1124749.

W. V. Baxter, J. Wendt, and M. C. Lin. IMPaSTo: A realistic, interactive model for paint. In S. N. Spencer, editor, *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering, NPAR*, pages 45–56. ACM, June 2004. doi:10.1145/987657.987665.

N. Ben-Zvi, J. Bento, M. Mahler, J. Hodgins, and A. Shamir. Line-drawing video stylization. *Computer Graphics Forum*, 2015. doi:10.1111/cgf.12729.

P. Bénard, F. Cole, A. Golovinskiy, and A. Finkelstein. Self-similar texture for coherent line stylization. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '10, pages 91–97. ACM, 2010. doi:10.1145/1809939.1809950.

P. Bénard, A. Bousseau, and J. Thollot. State-of-the-art report on temporal coherence for stylized animations. *Computer Graphics Forum*, 30(8):2367–2386, 2011. doi:10.1111/j.1467-8659.2011.02075.x.

P. Bénard, J. Lu, F. Cole, A. Finkelstein, and J. Thollot. Active strokes: Coherent line stylization for animated 3d models. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR '12, pages 37–46. Eurographics Association, 2012.

P. Bénard, F. Cole, M. Kass, I. Mordatch, J. Hegarty, M. S. Senn, K. Fleischer, D. Pesare, and K. Breeden. Stylizing animation by example. *ACM Trans. Graph.*, 32(4), July 2013. doi:10.1145/2461912.2461929.

P. Bénard, A. Hertzmann, and M. Kass. Computing smooth surface contours with accurate topology. *ACM Trans. Graph.*, 33(2):19:1–19:21, 2014. doi:10.1145/2558307.

F. Benichou and G. Elber. Output sensitive extraction of silhouettes from polygonal geometry. In *Computer Graphics and Applications, 1999. Proceedings. Seventh Pacific Conference on*, pages 60–69, 1999. doi:10.1109/PCCGA.1999.803349.

C. Benthin, S. Woop, M. Nießner, K. Selgrad, and I. Wald. Efficient ray tracing of subdivision surfaces using tessellation caching. In *Proceedings of the 7th High-Performance Graphics Conference*. ACM, 2015.

J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979. ISSN 0018-9340. doi:10.1109/TC.1979.1675432.

J. Bigler, J. Guilkey, C. Gribble, C. Hansen, and S. G. Parker. A Case Study: Visualizing Material Point Method Data. In *Eurographics /IEEE VGTC Symposium on Visualization*. The Eurographics Association, 2006. ISBN 3-905673-31-2. doi:10.2312/VisSym/EuroVis06/299-306.

BlenderNPR. Edge node v1.2.4. http://blendernpr.org/edge-node-v1-2-4-july-2015/, 7 2015a.

BlenderNPR. Solidify modifier contour/outline. http://blendernpr.org/solidify-modifier-contouroutline/, 2 2015b.

J. F. Blinn. A scan line algorithm for displaying parametrically defined surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 27–. ACM, 1978. doi:10.1145/800248.807364.

P. Bo, G. Luo, and K. Wang. A graph-based method for fitting planar b-spline curves with intersections. *Journal of Computational Design and Engineering*, 3(1):14 – 23, 2016. ISSN 2288-4300. doi:10.1016/j.jcde.2015.05.001.

L. Bourdev. Rendering nonphotorealistic strokes with temporal and arc-length coherence. Master's thesis, 1998. URL http://www.cs.brown.edu/research/graphics/art/bourdev-thesis.pdf.

S. Brabec and H.-P. Seidel. Shadow volumes on programmable graphics hardware. *Computer Graphics Forum*, 22(3):433–440, 2003. ISSN 1467-8659. doi:10.1111/1467-8659.00691.

D. Bremer and J. F. Hughes. Rapid approximate silhouette rendering of implicit surfaces. In *Proceesings of Implicit Surfaces 98*, 1998.

S. Breslav, K. Szerszen, L. Markosian, P. Barla, and J. Thollot. Dynamic 2d patterns for shading 3d scenes. *ACM Trans. Graph.*, 26(3), July 2007. ISSN 0730-0301. doi:10.1145/1276377.1276402.

B. Buchholz, N. Faraj, S. Paris, E. Eisemann, and T. Boubekeur. Spatio-temporal analysis for parameterizing animated lines. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 85–92. ACM, 2011. doi:10.1145/2024676.2024690.

M. Burns, J. Klawe, S. Rusinkiewicz, A. Finkelstein, and D. DeCarlo. Line drawings from volume data. *ACM Trans. Graph.*, 24(3):512–518, 2005. doi:10.1145/1073204.1073222.

S. Busking, A. Vilanova, and J. J. van Wijk. Particle-based non-photorealistic volume visualization. *The Visual Computer*, 24(5):335–346, 2008. ISSN 1432-2315. doi:10.1007/s00371-007-0192-x.

S. Campagna, L. Kobbelt, and H.-P. Seidel. Directed edges — a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–11, 1998. doi:10.1080/10867651.1998.10487494.

D. Card and J. L. Mitchell. Non-photorealistic rendering with pixel and vertex shaders. In *In Direct3D ShaderX, Wordware*, pages 319–333. Wordware Publishing, Inc, 2002.

L. Cardona and S. Saito. Hybrid-space localized stylization method for view-dependent lines extracted from 3d models. In *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering*, NPAR '15, pages 79–89. Eurographics Association, 2015.

L. Cardona and S. Saito. Temporally coherent and artistically intended stylization of feature lines extracted from 3d models. *Comput. Graph. Forum*, 35(7):137–146, Oct. 2016. ISSN 0167-7055. doi:10.1111/cgf.13011.

E. Catmull and J. H. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350 – 355, 1978. ISSN 0010-4485. doi:10.1016/0010-4485(78)90110-0.

D. Chen, Y. Zhang, H. Liu, and P. Xu. Real-time artistic silhouettes rendering for 3d models. In *8th International Symposium on Computational Intelligence and Design*, volume 1, pages 494–498, 2015a. doi:10.1109/ISCID.2015.201.

Z. Chen, B. Kim, D. Ito, and H. Wang. Wetbrush: Gpu-based 3d painting simulation at the bristle level. *ACM Trans. Graph.*, 34(6):200:1–200:11, Oct. 2015b. ISSN 0730-0301. doi:10.1145/2816795.2818066.

A. I. Choudhury and S. G. Parker. Ray tracing npr-style feature lines. In *NPAR '09: Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering*, pages 5–14. ACM, 2009. doi:10.1145/1572614.1572616.

R. Cipolla and P. Giblin. *Visual Motion of Curves and Surfaces*. Cambridge University Press, 2000. ISBN 0-521-63251-X.

F. Cole and A. Finkelstein. Partial visibility for stylized lines. In *Proceedings of the 6th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '08, pages 9–13. ACM, 2008. ISBN 978-1-60558-150-7. doi:10.1145/1377980.1377985.

F. Cole and A. Finkelstein. Two fast methods for high-quality line visibility. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):707–717, 2010. doi:10.1109/TVCG.2009.102.

F. Cole, A. Golovinskiy, A. Limpaecher, H. S. Barros, A. Finkelstein, T. Funkhouser, and S. Rusinkiewicz. Where do people draw lines? *ACM Trans. Graph.*, 27(3):88:1–88:11, 2008. doi:10.1145/1360612.1360687.

F. Cole, K. Sanik, D. DeCarlo, A. Finkelstein, T. Funkhouser, S. Rusinkiewicz, and M. Singh. How well do line drawings depict shape? *ACM Trans. Graph.*, 28(3):28:1–28:9, 2009. doi:10.1145/1531326.1531334.

F. Cole, M. Burns, K. Morley, A. Finkelstein, and P. Bénard. dpix. `https://gfx.cs.princeton.edu/proj/dpix/`, fork: `https://github.com/benardp/dpix`, 2010.

F. Cole, S. Rusinkiewicz, and D. DeCarlo. qrts, a port of the original "rtsc" software. `https://github.com/benardp/qrtsc`, 2011.

K. Crane, F. de Goes, M. Desbrun, and P. Schröder. Digital geometry processing with discrete exterior calculus. In *ACM SIGGRAPH 2013 courses*, SIGGRAPH '13. ACM, 2013. URL `https://www.cs.cmu.edu/~kmcrane/Projects/DDG/`.

B. Csébfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller. Fast visualization of object contours by non-photorealistic volume rendering. *Computer Graphics Forum*, 20(3): 452–460, 2001. doi:10.1111/1467-8659.00538.

C. J. Curtis. Loose and sketchy animation. In *ACM SIGGRAPH 98 Electronic Art and Animation Catalog*, SIGGRAPH '98, pages 145–. ACM, 1998. ISBN 1-58113-045-7. doi:10.1145/281388.281913.

C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer, and D. H. Salesin. Computer-generated watercolor. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 421–430. ACM, 1997. ISBN 0-89791-896-7. doi:10.1145/258734.258896.

B. R. de Araújo, D. S. Lopes, P. Jepp, J. A. Jorge, and B. Wyvill. A survey on implicit surface polygonization. *ACM Comput. Surv.*, 47(4):60:1–60:39, May 2015. ISSN 0360-0300. doi:10.1145/2732197.

D. DeCarlo. Depicting 3d shape using lines. In *Proc. SPIE*, volume 8291, pages 8291 – 8291 – 16, 2012. doi:10.1117/12.916463.

D. DeCarlo and S. Rusinkiewicz. Highlight lines for conveying shape. In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '07, pages 63–70. ACM, 2007. doi:10.1145/1274871.1274881.

D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella. Suggestive contours for conveying shape. *ACM Trans. Graph.*, 22(3):848–855, 2003. doi:10.1145/882262.882354.

D. DeCarlo, A. Finkelstein, and S. Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '04, pages 15–145. ACM, 2004. doi:10.1145/987657.987661.

P. Decaudin. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, 1996. URL http://phildec.users.sf.net/Research/RR-2919.php.

O. Deussen and T. Strothotte. Computer-generated pen-and-ink illustration of trees. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 13–18. ACM Press/Addison-Wesley Publishing Co., 2000. doi:10.1145/344779.344792.

D. P. Dobkin, A. R. Wilks, S. V. F. Levy, and W. P. Thurston. Contour tracing by piecewise linear approximations. *ACM Trans. Graph.*, 9(4):389–423, 1990. ISSN 0730-0301. doi:10.1145/88560.88575.

D. Dooley and M. F. Cohen. Automatic illustration of 3d geometric models: Lines. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, I3D '90, pages 77–82. ACM, 1990. doi:10.1145/91385.91422.

D. Ebert and P. Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of the Conference on Visualization '00*, VIS '00, pages 195–202. IEEE Computer Society Press, 2000. doi:10.1109/VISUAL.2000.885694.

E. Eisemann, H. Winnemöller, J. C. Hart, and D. Salesin. Stylized vector art from 3d models with region support. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, pages 1199–1207. Eurographics Association, 2008. doi:10.1111/j.1467-8659.2008.01258.x.

E. Eisemann, S. Paris, and F. Durand. A visibility algorithm for converting 3d meshes into editable 2d vector graphics. *ACM Trans. Graph.*, 28(3):83:1–83:8, 2009. ISSN 0730-0301. doi:10.1145/1531326.1531389.

G. Elber. Line art rendering via a coverage of isoparametric curves. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):231–239, 1995a. doi:10.1109/2945.466718.

G. Elber. Line illustrations $\in$ computer graphics. *The Visual Computer*, 11(6):290–296, 1995b. doi:10.1007/BF01898406.

G. Elber. Line art illustrations of parametric and implicit forms. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):71–81, 1998. ISSN 1077-2626. doi:10.1109/2945.675655.

G. Elber. The irit modeling environment. http://www.cs.technion.ac.il/~irit/, 3 2018.

G. Elber and E. Cohen. Hidden curve removal for free form surfaces. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 95–104. ACM, 1990. doi:10.1145/97879.97890.

G. Elber and E. Cohen. Probabilistic silhouette based importance toward line-art non-photorealistic rendering. *The Visual Computer*, 22(9):793–804, 2006. ISSN 1432-2315. doi:10.1007/s00371-006-0065-8.

G. Elber and M.-S. Kim. Geometric constraint solver using multivariate rational spline functions. In *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, SMA '01, pages 1–10. ACM, 2001. ISBN 1-58113-366-9. doi:10.1145/376957.376958.

H. Farid and E. P. Simoncelli. *Optimally rotation-equivariant directional derivative kernels*, pages 207–214. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-69556-1. doi:10.1007/3-540-63460-6_119.

J.-D. Favreau, F. Lafarge, and A. Bousseau. Fidelity vs. simplicity: A global approach to line drawing vectorization. *ACM Trans. Graph.*, 35(4):120:1–120:10, 2016. ISSN 0730-0301. doi:10.1145/2897824.2925946.

J. Fišer, O. Jamriška, M. Lukáč, E. Shechtman, P. Asente, J. Lu, and D. Sýkora. Stylit: Illumination-guided example-based stylization of 3d renderings. *ACM Trans. Graph.*, 35 (4):92:1–92:11, July 2016. ISSN 0730-0301. doi:10.1145/2897824.2925948.

E. Fogel, D. Halperin, and R. Wein. *CGAL Arrangements and Their Applications: A Step-by-Step Guide*. Springer Publishing Company, Incorporated, 2012. ISBN 3642172822.

K. Foster, P. Jepp, B. Wyvill, M. Sousa, C. Galbraith, and J. Jorge. Pen-and-ink for blobtree implicit models. *Computer Graphics Forum*, 24(3):267–276, 2005. doi:10.1111/j.1467-8659.2005.00851.x.

K. Foster, M. C. Sousa, F. F. Samavati, and B. Wyvill. Polygonal silhouette error correction: a reverse subdivision approach. *International Journal of Computational Science and Engineering*, 3(1):53–70, 2007. ISSN 1742-7185. doi:10.1504/IJCSE.2007.014465.

H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '80, pages 124–133. ACM, 1980. ISBN 0-89791-021-4. doi:10.1145/800250.807481.

M. Gangnet, J.-C. Hervé, T. Pudet, and J.-M. van Thong. Incremental computation of planar maps. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 345–354. ACM, 1989. ISBN 0-89791-312-4. doi:10.1145/74333.74369.

M. Gerl and T. Isenberg. Interactive example-based hatching. *Computers & Graphics*, 37 (1–2):65–80, 2013. doi:10.1016/j.cag.2012.11.003.

M. Glisse. An upper bound on the average size of silhouettes. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, pages 105–111. ACM, 2006. ISBN 1-59593-340-9. doi:10.1145/1137856.1137874.

E. H. Gombrich. *Art and Illusion: A Study in the Psychology of Pictorial Representation*. Princeton University Press, 2nd edition, 1961.

A. Gooch. Interactive non-photorealistic technical illustration. Master's thesis, Dept. of Computer Science, University of Utah, 1998.

B. Gooch, P.-P. J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, I3D '99, pages 31–38, New York, NY, USA, 1999. ACM. doi:10.1145/300523.300526.

N. Goodman. *Languages of Art: An Approach to a Theory of Symbols*. The Bobbs-Merrill Company, Inc., 1968.

T. Goodwin, I. Vollick, and A. Hertzmann. Isophote distance: A shading approach to artistic stroke thickness. In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '07, pages 53–62. ACM, 2007. doi:10.1145/1274871.1274880.

S. Grabli, F. Durand, and S. F. X. Density measure for line-drawing simplification. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 309–318, 2004. doi:10.1109/PCCGA.2004.1348362.

S. Grabli, E. Turquin, F. Durand, and F. X. Sillion. Programmable rendering of line drawing from 3d scenes. *ACM Trans. Graph.*, 29(2):18:1–18:20, 2010. doi:10.1145/1731047.1731056.

C. S. Haase and G. W. Meyer. Modeling pigmented materials for realistic image synthesis. *ACM Trans. Graph.*, 11(4):305–335, Oct. 1992. ISSN 0730-0301. doi:10.1145/146443.146452.

P. Haeberli. Paint by numbers: Abstract image representations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 207–214. ACM, 1990. ISBN 0-89791-344-2. doi:10.1145/97879.97902.

P. Hermosilla and P. P. Vázquez. Single pass gpu stylized edges. In *IV Iberoamerican Symposium in Computer Graphics*, pages 47–54, 2009.

A. Hertzmann. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In S. Green, editor, *ACM SIGGRAPH 99 Course Notes. Course on Non-Photorelistic Rendering*. ACM Press, 1999. URL `http://mrl.nyu.edu/publications/npr-course1999/`.

A. Hertzmann. Non-photorealistic rendering and the science of art. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '10, pages 147–157. ACM, 2010. ISBN 978-1-4503-0125-1. doi:10.1145/1809939.1809957.

A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000. doi:10.1145/344779.345074.

A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin. Image analogies. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 327–340. ACM, 2001. ISBN 1-58113-374-X. doi:10.1145/383259.383295.

A. Hertzmann, N. Oliver, B. Curless, and S. M. Seitz. Curve analogies. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 233–246. Eurographics Association, 2002. ISBN 1-58113-534-3. URL http://dl.acm.org/citation.cfm?id=581896.581926.

C. Hornung, W. Lellek, P. Rehwald, and W. Strasser. An area-oriented analytical visibility method for displaying parametrically defined tensor-product surfaces. *Computer Aided Geometric Design*, 2(1-3):197–205, 1985.

E. G. Houghton, R. F. Emnett, J. D. Factor, and C. L. Sabharwal. Implementation of a divide-and-conquer method for intersection of parametric surfaces. *Comput. Aided Geom. Des.*, 2(1-3):173–183, Sept. 1985. ISSN 0167-8396. doi:10.1016/0167-8396(85)90022-6.

S. C. Hsu and I. H. H. Lee. Drawing and animation using skeletal strokes. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 109–118, New York, NY, USA, 1994. ACM. doi:10.1145/192161.192186.

M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. *Chapter 39, Volume Rendering Techniques*, pages 667–692. Addison Wesley, 2004.

V. Interrante, H. Fuchs, and S. Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *Visualization, 1995. Visualization '95. Proceedings., IEEE Conference on*, pages 52–59, 438, 1995. doi:10.1109/VISUAL.1995.480795.

T. Isenberg, N. Halper, and T. Strothotte. Stylizing silhouettes at interactive rates: From silhouette edges to silhouette strokes. In *Computer Graphics Forum*, volume 21, pages 249–258, 2002. doi:10.1111/1467-8659.00584.

K. Jeong, A. Ni, S. Lee, and L. Markosian. Detail control in line drawings of 3d meshes. *The Visual Computer*, 21(8):698–706, 2005. doi:10.1007/s00371-005-0323-1.

T. Judd, F. Durand, and E. Adelson. Apparent ridges for line drawing. *ACM Trans. Graph.*, 26(3), 2007. doi:10.1145/1276377.1276401.

R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, and A. Finkelstein. Wysiwyg npr: Drawing strokes directly on 3d models. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 755–762. ACM, 2002. doi:10.1145/566570.566648.

R. D. Kalnins, P. L. Davidson, L. Markosian, and A. Finkelstein. Coherent stylized silhouettes. *ACM Trans. Graph.*, 22(3):856–861, 2003. doi:10.1145/882262.882355.

R. D. Kalnins, P. L. Davidson, and D. M. Bourguignon. Jot. http://jot.cs.princeton.edu, fork: https://github.com/benardp/jot-lib, 2007.

E. Kalogerakis, D. Nowrouzezahrai, S. Breslav, and A. Hertzmann. Learning hatching for pen-and-ink illustration of surfaces. *ACM Trans. Graphics*, 31(1), 2012.

M. Kaplan. Hybrid quantitative invisibility. In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '07, pages 51–52. ACM, 2007. ISBN 978-1-59593-624-0. doi:10.1145/1274871.1274879.

K. Karsch and J. C. Hart. Snaxels on a plane. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 35–42. ACM, 2011. doi:10.1145/2024676.2024683.

M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1988. ISSN 1573-1405. doi:10.1007/BF00133570.

A. Kaufman and K. Mueller. *Overview of Volume Rendering*, volume 7, pages 127 – 174. Butterworth-Heinemann, 2005. ISBN 9780123875822. doi:10.1016/B978-012387582-2/50009-5.

L. Kettner and E. Welzl. *Contour Edge Analysis for Polyhedron Projections*, pages 379–394. Springer Berlin Heidelberg, 1997. ISBN 978-3-642-60607-6. doi:10.1007/978-3-642-60607-6_25.

K.-J. Kim and N. Baek. Fast extraction of polyhedral model silhouettes from moving viewpoint on curved trajectory. *Computers & Graphics*, 29(3):393–402, 2005. ISSN 0097-8493. doi:10.1016/j.cag.2005.03.009.

G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Moller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 67–. IEEE Computer Society, 2003. doi:10.1109/VISUAL.2003.1250414.

D. Kirsanov, P. V. Sander, and S. J. Gortler. Simple silhouettes for complex surfaces. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, pages 102–106. Eurographics Association, 2003. ISBN 1-58113-687-0.

A. W. Klein, W. Li, M. M. Kazhdan, W. T. Corrêa, A. Finkelstein, and T. A. Funkhouser. Non-photorealistic virtual environments. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 527–534. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 1-58113-208-5. doi:10.1145/344779.345075.

L. P. Kobbelt, K. Daubert, and H.-P. Seidel. Ray tracing of subdivion surfaces. In *Rendering Techniques*, 1998.

J. J. Koenderink. What does the occluding contour tell us about solid shape? *Perception*, 13 (3):321–330, 1984. doi:10.1068/p130321.

J. J. Koenderink and A. J. van Doorn. The shape of smooth objects and the way contours end. *Perception*, 11(2):129–137, 1982. doi:10.1068/p110129.

A. Kolliopoulos, J. M. Wang, and A. Hertzmann. Segmentation-based 3d artistic rendering. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR '06, pages 361–370. Eurographics Association, 2006. ISBN 3-905673-35-5. doi:10.2312/EGWR/EGSR06/361-370.

M. Kolomenkin, I. Shimshoni, and A. Tal. Demarcating curves for shape illustration. *ACM Trans. Graph.*, 27(5):157:1–157:9, 2008. doi:10.1145/1409060.1409110.

P. Kubelka. New contributions to the optics of intensely light-scattering materials. part i. *Josa*, 38(5):448–457, 1948.

A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering*, NPAR '00, pages 13–20. ACM, 2000. doi:10.1145/340916.340918.

K. Lang and M. Alexa. The markov pen: Online synthesis of free-hand drawing styles. In *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering*, NPAR '15, pages 203–215. Eurographics Association, 2015.

K. Lawonn, I. Viola, B. Preim, and T. Isenberg. A survey of surface-based illustrative rendering for visualization. *Computer Graphics Forum*, 2018. ISSN 1467-8659. doi:10.1111/cgf.13322.

Y. Lee, L. Markosian, S. Lee, and J. F. Hughes. Line drawings via abstracted shading. *ACM Trans. Graph.*, 26(3), 2007. doi:10.1145/1276377.1276400.

W. Leister. Computer generated copper plates. In *Computer Graphics Forum*, volume 13, pages 69–77. Wiley Online Library, 1994.

B. Lichtenbelt, R. Crane, and S. Naqvi. *Introduction to Volume Rendering*. Prentice-Hall, Inc., 1998. ISBN 0-13-861683-3.

C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, Department of Mathematics, The University of Utah, January 1987.

W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169. ACM, 1987. ISBN 0-89791-227-6. doi:10.1145/37401.37422.

J. Lu, C. Barnes, S. DiVerdi, and A. Finkelstein. Realbrush: Painting with examples of physical media. *ACM Trans. Graph.*, 32(4):117:1–117:12, July 2013. ISSN 0730-0301. doi:10.1145/2461912.2461998.

J. Lu, C. Barnes, C. Wan, P. Asente, R. Mech, and A. Finkelstein. Decobrush: Drawing structured decorative patterns by example. *ACM Trans. Graph.*, 33(4):90:1–90:9, July 2014. ISSN 0730-0301. doi:10.1145/2601097.2601190.

E. B. Lum and K.-L. Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '02, pages 67–ff. ACM, 2002. doi:10.1145/508530.508542.

P. Mamassian and M. S. Landy. Observer biases in the 3d interpretation of line drawings. *Vision Research*, 38:2817—2832, 1998.

L. Markosian, M. A. Kowalski, D. Goldstein, S. J. Trychin, J. F. Hughes, and L. D. Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th Annual Conference on*

Computer Graphics and Interactive Techniques, SIGGRAPH '97, pages 415–420. ACM Press/Addison-Wesley Publishing Co., 1997. doi:10.1145/258734.258894.

D. Marr. Analysis of occluding contour. *Proceedings of the Royal Society of London B: Biological Sciences*, 197(1129):441–475, 1977. ISSN 0080-4649. doi:10.1098/rspb.1977.0080.

D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London B: Biological Sciences*, 207(1167):187–217, 1980.

M. Masuch, S. Schlechtweg, and B. Schönwälder. dali! - drawing animated lines! In *In Proceedings of Simulation und Animation '97, SCS Europe*, pages 87–96, 1997.

N. Max. Weights for computing vertex normals from facet normals. *Journal of Graphics Tools*, 4(2):1–6, 1999. ISSN 1086-7651. doi:10.1080/10867651.1999.10487501.

M. McGuire. Observations on silhouette sizes. *jgt*, 9(1):1–12, 2004. URL `http://www.cs.brown.edu/research/graphics/games/SilhouetteSize/index.html`.

M. McGuire and J. F. Hughes. Hardware-determined feature edges. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '04, pages 35–47. ACM, 2004. doi:10.1145/987657.987663.

B. J. Meier. Painterly rendering for animation. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 477–484. ACM, 1996. ISBN 0-89791-746-4. doi:10.1145/237170.237288.

M. D. Meyer, P. Georgel, and R. T. Whitaker. Robust particle systems for curvature dependent sampling of implicit surfaces. In *Proceedings of the International Conference on Shape Modeling and Applications 2005*, SMI '05, pages 124–133. IEEE Computer Society, 2005. ISBN 0-7695-2379-X. doi:10.1109/SMI.2005.41.

J. Mitchell, M. Francke, and D. Eng. Illustrative rendering in team fortress 2. In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '07, pages 71–76. ACM, 2007. ISBN 978-1-59593-624-0. doi:10.1145/1274871.1274883.

Z. Nagy, J. Schneider, and R. Westermann. Interactive volume illustration. In *Vision, Modeling and Visualization 2002*, Nov. 2002.

A. Ni, K. Jeong, S. Lee, and L. Markosian. Multi-scale line drawings from 3d meshes. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 133–137. ACM, 2006. doi:10.1145/1111411.1111435.

M. Nienhaus and J. Döllner. Blueprints: Illustrating architecture and technical parts using hardware-accelerated non-photorealistic rendering. In *Proceedings of Graphics Interface 2004*, GI '04, pages 49–56. Canadian Human-Computer Communications Society, 2004.

M. Nießner, C. Loop, M. Meyer, and T. Derose. Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Trans. Graph.*, 31(1):6:1–6:11, Feb. 2012. ISSN 0730-0301. doi:10.1145/2077341.2077347.

J. D. Northrup and L. Markosian. Artistic silhouettes: A hybrid approach. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering*, NPAR '00, pages 31–37. ACM, 2000. doi:10.1145/340916.340920.

P. O'Donovan and A. Hertzmann. Anipaint: interactive painterly animation from video. *IEEE transactions on visualization and computer graphics*, 18(3):475–87, mar 2012. ISSN 1941-0506. doi:10.1109/TVCG.2011.51.

S. Ogaki and I. Georgiev. Production ray tracing of feature lines. In *SIGGRAPH Asia 2018 Technical Briefs*, SA '18, pages 15:1–15:4. ACM, 2018. ISBN 978-1-4503-6062-3. doi:10.1145/3283254.3283273.

Y. Ohtake, A. Belyaev, and H.-P. Seidel. Ridge-valley lines on meshes via implicit surface fitting. *ACM Trans. Graph.*, 23(3):609–612, 2004. doi:10.1145/1015706.1015768.

M. Olson and H. Zhang. Silhouette extraction in hough space. *Computer Graphics Forum*, 25(3):273–282, 2006. doi:10.1111/j.1467-8659.2006.00946.x.

M. Pauly, R. Keiser, and M. Gross. Multi-scale feature extraction on point-sampled surfaces. *Computer Graphics Forum*, 22(3):281–289, 2003. doi:10.1111/1467-8659.00675.

H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 335–342. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 1-58113-208-5. doi:10.1145/344779.344936.

M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 3rd edition, 2016. ISBN 0128006455.

Pixar. Opensubdiv. `http://graphics.pixar.com/opensubdiv`, 2019.

S. Plantinga and G. Vegter. Computing contour generators of evolving implicit surfaces. *ACM Trans. Graph.*, 25(4):1243–1280, 2006. doi:10.1145/1183287.1183288.

M. Pop, C. Duncan, G. Barequet, M. Goodrich, W. Huang, and S. Kumar. Efficient perspective-accurate silhouette computation and applications. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry*, SCG '01, pages 60–68. ACM, 2001. doi:10.1145/378583.378618.

E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 581–. ACM, 2001. ISBN 1-58113-374-X. doi:10.1145/383259.383328.

B. Preim and T. Strothotte. Tuning rendered line-drawings. In *WSCG'95*, pages 228–238, 1995.

R. Raskar. Hardware support for non-photorealistic rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '01, pages 41–47. ACM, 2001. doi:10.1145/383507.383525.

R. Raskar and M. Cohen. Image precision silhouette edges. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, I3D '99, pages 135–140. ACM, 1999. doi:10.1145/300523.300539.

P. Rideout. Silhouette extraction, 2010. http://prideout.net/blog/?p=54.

L. Roberts. *Machine Perception of Three-Dimensional Solids*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1963.

S. Roettger. The volume library. http://schorsch.efi.fh-nuernberg.de/data/volume/, 2012.

P. Rosin and J. Collomosse. *Image and Video-Based Artistic Stylisation*. Springer, 2013.

J. R. Rossignac and M. van Emmerik. Hidden contours on a frame-buffer. In *Proceedings of the Seventh Eurographics Conference on Graphics Hardware*, EGGH'92, pages 188–203. Eurographics Association, 1992. doi:10.2312/EGGH/EGGH92/188-203.

S. Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *Proceedings of the 3D Data Processing, Visualization, and Transmission, 2Nd International Symposium*, 3DPVT '04, pages 486–493. IEEE Computer Society, 2004. doi:10.1109/3DPVT.2004.54.

S. Rusinkiewicz, F. Cole, D. DeCarlo, and A. Finkelstein. Line drawings from 3d models. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, pages 39:1–39:356, New York, NY, USA, 2008. ACM. doi:10.1145/1401132.1401188.

B. Sabiston. Waking life: Making of, 2001. URL http://www.flatblackfilms.com/Flat_Black_Films/Rotoshop.html.

T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 197–206. ACM, 1990. doi:10.1145/97879.97901.

P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 327–334. ACM Press/Addison-Wesley Publishing Co., 2000. doi:10.1145/344779.344935.

P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 27(5):144:1–144:9, 2008. doi:10.1145/1409060.1409097.

B. Sayim and P. Cavanagh. What line drawings reveal about the visual brain. *Frontiers in Human Neuroscience*, 5:118, 2011. ISSN 1662-5161. doi:10.3389/fnhum.2011.00118.

S. Schein and G. Elber. Adaptive extraction and visualization of silhouette curves from volumetric datasets. *Vis. Comput.*, 20(4):243–252, 2004. doi:10.1007/s00371-003-0230-2.

R. Schmidt. Shapeshop. http://www.shapeshop3d.com/, 2008.

R. Schmidt, T. Isenberg, P. Jepp, K. Singh, and B. Wyvill. Sketching, scaffolding, and inking: A visual history for interactive 3d modeling. In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '07, pages 23–32. ACM, 2007. ISBN 978-1-59593-624-0. doi:10.1145/1274871.1274875.

J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, Oct. 1997.

M. Singh and S. Schaefer. Suggestive hatching. In *Proc. Computational Aesthetics*, 2010.

P.-P. J. Sloan, W. Martin, A. Gooch, and B. Gooch. The lit sphere: A model for capturing npr shading from art. In *Proceedings of Graphics Interface 2001*, GI '01, pages 143–150. Canadian Information Processing Society, 2001. ISBN 0-9688808-0-0.

M. C. Sousa and J. W. Buchanan. Computer-generated graphite pencil rendering of 3d polygonal models. *Computer Graphics Forum*, 18(3):195–208, 1999. doi:10.1111/1467-8659.00340.

M. C. Sousa and P. Prusinkiewicz. A Few Good Lines: Suggestive Drawing of 3D Models. *Computer Graphics Forum*, 2003. doi:10.1111/1467-8659.00685.

J.-F. St-Amour. The illustrative rendering of prince of persia. In *ACM SIGGRAPH 2010 Courses*, 2010. URL http://www.cs.williams.edu/~morgan/SRG10.

J. Stam. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 395–404. ACM, 1998a. ISBN 0-89791-999-8. doi:10.1145/280814.280945.

J. Stam. Evaluation of loop subdivision surfaces, 1998b.

M. Stich, C. Wächter, and A. Keller. *Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders*, chapter 11. Addison-Wesley Professional, 2007. ISBN 9780321545428.

M. Stroila, E. Eisemann, and J. Hart. Clip art rendering of smooth isosurfaces. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):135–145, 2008. doi:10.1109/TVCG.2007.1058.

M. Stroila, E. Bachta, W. Jarosz, W. Su, and J. Hart. Wickbert. fork: https://github.com/benardp/Wickbert, 11 2011.

T. Tejima, M. Fujita, and T. Matsuoka. Direct ray tracing of full-featured subdivision surfaces with bezier clipping. *Journal of Computer Graphics Techniques (JCGT)*, 4(1):69–83, March 2015. ISSN 2331-7418. URL http://jcgt.org/published/0004/01/04/.

A. Thibault and S. Cavanaugh. Making concept art real for borderlands. In *ACM SIGGRAPH 2010 Courses*, 2010. URL http://www.cs.williams.edu/~morgan/SRG10.

J.-P. Thirion and A. Gourdon. The 3d marching lines algorithm. *Graphical Models and Image Processing*, 58(6):503 – 509, 1996. doi:10.1006/gmip.1996.0042.

D. Vanderhaeghe, R. Vergne, P. Barla, and W. Baxter. Dynamic Stylized Shading Primitives. In *NPAR '11:Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, pages 99–104, Aug. 2011. doi:10.1145/2024676.2024693.

L. Váša, P. Vaněček, M. Prantl, V. Skorkovská, P. Martínek, and I. Kolingerová. Mesh statistics for robust curvature estimation. In *Proceedings of the Symposium on Geometry Processing*, SGP'16, pages 271–280. Eurographics Association, 2016. doi:10.1111/cgf.12982.

R. Vergne, D. Vanderhaeghe, J. Chen, P. Barla, X. Granier, and C. Schlick. Implicit Brushes for Stylized Line-based Rendering. *Computer Graphics Forum*, 30(2):513–522, 2011. doi:10.1111/j.1467-8659.2011.01892.x.

M. Webb, E. Praun, A. Finkelstein, and H. Hoppe. Fine tone control in hardware hatching. In *Proceedings of the 2Nd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '02, pages 53–ff. ACM, 2002. ISBN 1-58113-494-0. doi:10.1145/508530.508540.

H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984. ISSN 0730-0301. doi:10.1145/357332.357335.

R. A. Weiss. Be vision, a package of ibm 7090 fortran programs to draw orthographic views of combinations of plane and quadric surfaces. *Journal of the ACM*, 13(2):194–204, 1966. ISSN 0004-5411. doi:10.1145/321328.321330.

B. Whited, E. Daniels, M. Kaschalk, P. Osborne, and K. Odermatt. Computer-assisted animation of line and paint in disney's paperman. In *Proc. SIGGRAPH Talks*. ACM, 2012. ISBN 978-1-4503-1683-5. doi:10.1145/2343045.2343071.

J. Willats. *Art and Representation: New Princples in the Analysis of Pictures*. Princeton University Press, 1997.

J. Willats and F. Durand. Defining pictorial style: Lessons from linguistics and computer graphics. *Axiomathes*, 15(3):319–351, 2005. doi:10.1007/s10516-004-5449-7.

B. Wilson and K.-L. Ma. Rendering complexity in computer-generated pen-and-ink illustrations. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '04, pages 129–137. ACM, 2004. ISBN 1-58113-887-3. doi:10.1145/987657.987674.

G. Winkenbach and D. H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 91–100. ACM, 1994. doi:10.1145/192161.192184.

G. Winkenbach and D. H. Salesin. Rendering parametric surfaces in pen and ink. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 469–476. ACM, 1996. doi:10.1145/237170.237287.

H. Winnemöller, J. E. Kyprianidis, and S. C. Olsen. Xdog: An extended difference-of-gaussians compendium including advanced image stylization. *Computers & Graphics*, 36 (6):740 – 753, 2012. ISSN 0097-8493. doi:10.1016/j.cag.2012.03.004.

A. P. Witkin and P. S. Heckbert. Using particles to sample and control implicit surfaces. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 269–277. ACM, 1994. ISBN 0-89791-667-0. doi:10.1145/192161.192227.

X. Xie, Y. He, F. Tian, H.-S. Seah, X. Gu, and H. Qin. An effective illustrative visualization framework based on photic extremum lines (PELs). *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1328–1335, 2007. doi:10.1109/TVCG.2007.70538.

H. Xu, M. X. Nguyen, X. Yuan, and B. Chen. Interactive silhouette rendering for point-based models. In *Proceedings of the First Eurographics Conference on Point-Based Graphics*, SPBG'04, pages 13–18. Eurographics Association, 2004. doi:10.2312/SPBG/SPBG04/013-018.

C. I. Yessios. Computer drafting of stones, wood, plant and ground materials. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, pages 190–198. ACM, 1979. ISBN 0-89791-004-4. doi:10.1145/800249.807443.

S. Yoshizawa, A. Belyaev, H. Yokota, and H.-P. Seidel. Fast and faithful geometric algorithm for detecting crest lines on meshes. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, PG '07, pages 231–237. IEEE Computer Society, 2007. doi:10.1109/PG.2007.24.

L. Zhang, Y. He, X. Xie, and W. Chen. Laplacian lines for real-time shape illustration. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 129–136. ACM, 2009. doi:10.1145/1507149.1507170.

L. Zhang, J. Xia, X. Ying, Y. He, W. Mueller-Wittig, and H.-S. Seah. Efficient and robust 3d line drawings using difference-of-gaussian. *Graphical Models*, 74(4):87 – 98, 2012. doi:10.1016/j.gmod.2012.03.006.

L. Zhang, Q. Sun, and Y. He. Splatting lines: An efficient method for illustrating 3d surfaces and volumes. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 135–142. ACM, 2014. doi:10.1145/2556700.2556703.

D. Zorin and P. Schröder. Subdivision for modeling and animation. In *ACM SIGGRAPH 2000 Courses*, 2000. URL `https://mrl.nyu.edu/publications/subdiv-course2000/`.